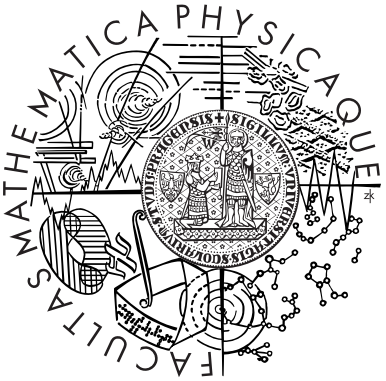
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Michal Kebrt

Unit Checking for Java IDE

Department of Software Engineering
Advisor: RNDr. Ondřej Šerý
Study Program: Computer Science, Software Systems

Prague, July 2009

First of all, I would like to thank my advisor Ondřej Šerý for his suggestions, ideas, and a helpful attitude for all the time I was working on this thesis.

Many thanks go to the NASA Ames Research Center, especially to Peter Mehlitz, for providing a lot of valuable information about Java PathFinder.

Last but not least, I would like to thank my friends and family for their support in my life and studies.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

Prague, 30th July 2009

_____

Michal Kebrt

Contents

List of Figures

Listings

Název práce: Unit Checking for Java IDE
Autor: Michal Kebrt
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Ondřej Šerý
E-mail vedoucího: ondrej.sery@dsrg.mff.cuni.cz
Abstrakt: Model checking programů představuje rychle se rozvíjející oblast výzkumu. Bohužel tato technika zatím není kromě několika speciálních případů (např. verifikace ovladačů pomocí nástroje SLAM) příliš rozšířena ve standardním procesu vývoje software. Věříme ovšem, že by mohla nastat změna, pokud by vývojáři měli možnost používat nástroje z oblasti techniky model checking podobným způsobem jako používají nástroje pro testování.

Tato práce prezentuje nástroj UnitCheck, který doplňuje standardní testovaní Java tříd (unit testing) o metodu model checking. Vývojáři, kteří běžně využívají unit testování, mohou tento nástroj aplikovat na standardní testovací scénáře a díky části provádějící model checking (model checker) integrované v nástroji UnitCheck tak využít možnost průchodu celým stavovým prostorem daného testu. UnitCheck v sobě integruje dva dobře známé nástroje z oblasti programování v jazyce Java. JUnit je použit jako testovací framework, zatímco Java PathFinder umožňuje provádět model checking vstupních testů. Součástí celého nástroje je plugin pro prostředí Eclipse, který přehledně zobrazuje výsledky testů způsobem známým z unit testování a pokročilým uživatelům zároveň umožňuje prohlédnout si podrobnější informace. UnitCheck je také integrován do nástroje Ant, což umožňuje spouštět kontrolu testů (unit checking) ve standardním procesu vývoje a kompilace Java programů.

Klíčová slova: unit testování, model checking, JUnit, Java PathFinder, Eclipse

Title: Unit Checking for Java IDE
Author: Michal Kebrt
Department: Department of Software Engineering
Supervisor: RNDr. Ondřej Šerý
Supervisor's e-mail address: ondrej.sery@dsrg.mff.cuni.cz
Abstract: Code model checking is a rapidly advancing research topic. However, apart from very constrained scenarios (e.g., verification of device drivers by SLAM), the code model checking tools are not widely used in general software development process. We believe that this could be changed if the developers could use the tools in the same way they already use testing tools.

In this work, we present the UnitCheck tool, which enhances the standard unit testing of Java code with model checking. A developer familiar with unit testing can apply the tool on standard unit test scenarios and benefit from the exhaustive traversal performed by a code model checker, which is employed inside UnitCheck. Two well-known Java tools are integrated in UnitCheck. JUnit is used as a testing framework and Java PathFinder provides the model checking capability. The UnitCheck plugin for Eclipse presents the checking results in a convenient way known from unit testing, while providing also a verbose output for the expert users. The UnitCheck Ant task allows to incorporate unit checking in the standard Java development and build process.

Keywords: unit testing, model checking, JUnit, Java PathFinder, Eclipse

Chapter 1

Introduction

1.1 Motivation

In recent years, the field of code model checking has advanced significantly. There exist a number of code model checkers targeting mainstream programming languages such as C, Java, and C# (e.g., SLAM [27], CMBC [4], BLAST [23], Java PathFinder [11], and MoonWalker [35]). In spite of this fact, the adoption of the code model checking technologies in the industrial software development process is still very slow. This is caused by two main reasons:

- limited scalability to large software,
- missing tool-supported integration into the development process.

The current model checking tools can handle programs up to tens of KLOC and often require manual simplifications of the code under analysis [56]. Unfortunately, such program size is still several orders of magnitude smaller than the size of many industrial projects.

Apart from the scalability issues, there is virtually no support for integration of the code model checkers into the development process. Although some tools feature a user interface in the form of a plugin for a mainstream IDE (e.g., SATABS [32]), creation of a particular checking scenario is not often discussed or supported in any way. A notable exception is SLAM and its successful application in the very specific domain of kernel device drivers.

These two obstacles might be overcome by employing code model checking in a way similar to unit testing – we use the term *unit checking* first proposed in [38]. Unit testing is widely used technique for ensuring quality of software during its development and maintenance. Frameworks (e.g., JUnit [13]) exist that facilitate creation, execution, and evaluation of simple unit tests and developers are familiar with writing test suites. Providing model checking tools with a similar interface would allow developers to directly benefit from model checking technology (e.g., of exploration of all thread interleavings and random choices) without changing their habits. Moreover, applying model checking to smaller code units also helps avoiding the state explosion problem, the main issue of the model checking tools.

1.2 Goals

The goals of the presented thesis are to

- explore possibilities of unit checking using the Java PathFinder model checker,
- implement support for unit checking into a Java IDE (either NetBeans or Eclipse),
- consider extending JUnit to support unit checking.

1.3 Outline of the Thesis

Two techniques used for ensuring quality of software are described in Chapter 2, *Unit Testing and Model Checking*. This chapter also contains the list of existing integrations of unit testing and model checking. A novel idea of unit checking, i.e., running unit tests under a code model checker, is proposed and described in more detail. JUnit and Java PathFinder, the tools used in our integrated work, are also thoroughly presented.

The integration of JUnit and Java PathFinder is called UnitCheck. Chapter 3, *Unit Checking with JUnit and JPF*, shows the design of UnitCheck, how it uses extension interfaces provided by JUnit and JPF, and difficulties that had to be solved.

UnitCheck itself is a Java library with no user interface. Chapter 4, *UnitCheck in 3rd Party Programs*, explains how UnitCheck is embedded in third party programs, especially in Eclipse and Ant.

A short case study is presented in Chapter 5, *Case Study*. The program selected for the case study was earlier used in a contest for various verification tools.

A list of various related techniques and tools is presented in Chapter 6, *Related Work*. It includes testing frameworks and integrations of model checkers in IDEs.

The thesis is concluded in Chapter 7, *Summary and Conclusion*. A few ideas for future work are suggested.

Chapter 2

Unit Testing and Model Checking

This chapter focuses on the general description of two techniques used for the quality assurance of software. Section 2.1 describes unit testing, while Section 2.2 covers model checking. In each section, one particular example of a tool is taken in more detail – JUnit as a unit testing framework and Java PathFinder as a model checker. Section 2.3 lists different approaches for integrating unit testing and model checking. A brief introduction to UnitCheck, which integrates both previous tools, follows.

2.1 Unit Testing

In software development, testing is one of the important parts of the whole development process. Software testing includes many different methods and approaches. To make sure that the smallest pieces of a software program (or *units*) meet specification and work exactly as programmers expect, unit testing [22] is usually employed. In procedural languages, unit is a procedure (or function) or a set of procedures that make up a data type. In object-oriented world, units are methods or the whole classes. Instead of unit, the term *class under test* (CUT) is often used.

Unit tests (or *test cases*) are primarily used by programmers to feel confident about the code, which is in contrast to functional and acceptance tests that are prepared for the end users. Unit tests are usually written in the same language as the original code. Each test comprises of a *test driver* that executes the unit (e.g., a method is called on an instance of a class under test) and checks that it works correctly (e.g., the return value of the method meets the expectation). A lot testing frameworks for different programming languages exist that allow for easy creation and batch running of test cases. They are collectively known as xUnit frameworks – JUnit for Java and NUnit for .NET languages are two concrete examples.

In ideal situation, units are tested in isolation from other units. With growing complexity of units, it becomes hard to test them separately. Thus, programmers have to follow a set of rules [46] to write easy testable units. It includes the usage of stubbing and mock objects to implement a dummy environment of dependencies.

Unit testing, part of the *extreme programming* methodology, is often used together with *test-driven development* [50], where

- tests are written before the functional code is implemented,
- changes in the code are always initiated by a test fail (either by an existing test or a new one),
- after the code has been changed or added, all tests are run again to check that nothing has been broken by the changes,
- refactoring is frequently used to make the overall code quality better.

2.1.1 JUnit

JUnit [13, 64] is a framework for writing unit tests for Java classes in a convenient way. It was one of the first xUnit frameworks and now it is a very popular tool used in a variety of projects because of the features it provides:

- Various user interfaces for running tests – both command-line and graphical ones. JUnit is integrated into IDEs (Eclipse, NetBeans, ...) and other development tools (Ant, Maven, ...).
- A lot of extensions, which add more functionality to JUnit, exist (e.g., test coverage reports).

In Java applications, a method is often understood as a unit. Therefore, a unit test tries to show that the method behaves according to its API contract. Specifically, the test checks whether

- for a particular input data (method arguments), the method returns correct values,
- in the situations specified by the contract, the method throws correct exceptions.

So far, a couple of terms were used without a proper definition. Now it is time to make the things clear within the JUnit context.

Test (test method)
In JUnit, unit tests, or simply tests, are represented as Java methods. JUnit provides means for marking a method as a test.

Test case (test class)
More tests can be grouped within a test case. It is a Java class containing one or more test methods.

Test suite
Test suites stand at the top of the test hierarchy. Test cases and tests can be added into a test suite and run together.

Test runner
Runners allow to execute test suites (and test cases) under JUnit. Besides graphical and command-line runners bundled with JUnit, additional runners can be implemented.

The procedure of writing JUnit tests is not very complicated and JUnit IDE plugins can make it even easier. There is usually one test case for each Java class to be tested. Within the test case, there exist one or more tests for each method of the tested class.

JUnit now exists in two versions – 3 and 4. Although the version 4 was first released in 2006, the previous version is still very popular because the backward compatibility has been kept. The two versions are completely different in the way of creating tests [37]. Table 2.1 brings a brief comparison, the main change in the version 4 is the usage of Java annotations.

Table 2.1 Comparison of JUnit 3 and JUnit 4

	JUnit 3	**JUnit 4**
Test cases and tests	A test class extends `TestCase`, each test method's name must start with 'test'.	Test cases are not explicitly marked. Tests are marked with the `@Test` annotation.
Initialization and cleanup	JUnit invokes the `setUp` and `tearDown` methods before and after each test.	Methods marked with the `@Before` and `@After` annotations are invoked before and after each test.
Exceptions	If an exception is expected to be thrown by the code under test, the `fail` method is used when the exception is not thrown.	It is enough to use the `@Test(expected=<classname>)` annotation.
Other features	GUI test runners (Swing, AWT) are available (not in the version 4). JUnit distinguishes between failures and errors[1].	Annotations for omitting tests from execution (`@Ignore`) and for setting timeouts given to execution (`@Test(timeout=<time>`) are provided.

JUnit assertions are used to check the conditions that must be true in a successfully finished test. As an example, `assertEquals(Object, Object)` asserts that two objects are equal. If an assertion does not hold, an exception is thrown, caught by JUnit, and reported as a test failure.

Listing 2.1 shows a simple class with an implementation of a couple of math operations (addition, division). Below we see the test case for this class written in JUnit 4 notation (the `@Test` annotation is used before each method). Let us start with the first test. When the implementation of the `add` method does not work correctly (in other words, when 7+5 is not 12), the assert will throw an exception and JUnit will report a test failure. In the second test, an extended version of the `@Test` annotation is used. It says that a particular exception is expected to be thrown when the division by zero is performed. In contrast to the previous example, when the exception is not thrown, it means a test failure.

[1] *Failure* is anticipated, it is the result of a failed assertion in a test. *Error* is not anticipated, it is caused by an uncaught exception in a test.

Listing 2.1: Example of JUnit Test

```
public class SimpleMath {
  public static double add(double a, double b) {
    return a + b;
  }
  public static double divide(double a, double b) {
    if (b == 0) {
      throw new ArithmeticException("Division by zero");
    }
    return a / b;
  }
}
// ---------------------------------------------------------------------
public class SimpleMathTest {
  @Test
  public void testAddition() {
    assertEquals(12, SimpleMath.add(7, 5));
  }
  @Test(expected=ArithmeticException.class)
  public void testDivisionByZero() {
    SimpleMath.divide(1, 0);
  }
}
```

2.2 Model Checking

To ensure that software systems meet their specifications, a wide range of verification techniques is applied. Peer reviewing, static analysis of code, and testing are most often used. The previous section described unit testing – a kind of software testing. Another verification technique is *model checking* [36, 30, 52] used mainly for the verification of abstract representations of concurrent systems. In recent years, it has become more popular in the industry and was successfully employed for the verification of a couple of software and hardware projects [43, 42, 68].

The basic concept refers to mathematical logic. Model checking determines whether a given structure is a model of a given logical formula. When checking hardware and software systems, the terms model and property are often used instead of mathematically rigorous terms structure and formula.

- *Models* accurately describe the behavior of systems. Both finite and infinite systems are supported by various model checking tools. In either case, the *state* of the system during checking holds the current values of variables, program counter, etc. *Transitions* define how the system moves from one state to another. Models are written in special-purpose modelling languages (e.g., Promela [17]) or derived from general programming languages (e.g., old versions of JPF [11]).

- *Properties* are derived from system's specifications and tell what the system should do and what not (e.g., a property requires that a deadlock, uncaught exception, or null-pointer dereference does not occur in the system). Properties are usually

expressed as formulas in temporal logic (e.g., LTL in Spin [17]) or by general programming languages (e.g., Java in JPF [11]). Some properties are generic and they can be hard coded in model checkers (e.g., a deadlock property in Spin).

The main objective of model checking is to efficiently find subtle errors in models or prove they are errorfree. In other words, to check whether a set of properties hold in a model or not. The answer provided by model checkers is either 'yes' or 'no + counterexample'. A *counterexample* (or *error trace*) describes the complete execution path leading to the property violation.

Two basic model checking approaches exist. *Symbolic model checking* [49] represents sets of states and transition relations as boolean formulas. It uses algorithms that work with these symbolic representations and thereby tries to overcome the state explosion problem which will be described later. *Explicit-state model checking* represents each reachable state of the model explicitly and it typically stores the state in a hash table[2]. Explicit-state model checkers exhaustively explore the state space of the model not to miss any state, in which a property might fail.

The *state space explosion* problem is often discussed when dealing with model checking. The number of states in a model is exponential in the size of its description. It is very memory-consuming to store all the states and very time-consuming to traverse them. Although the growth in computer resources allows to check bigger systems than a decade ago, still the range of systems that can be checked is quite narrow. There are a number of approaches to tackle this problem and make model checkers more scalable. Some of these approaches will be pointed out in the next section.

It is important to note that the verification is done with a model of the system, not with the actual system. Therefore, the results of the verification process are the more relevant, the better the model resembles the actual system. As a consequence, the analysis of results provided by a model checker leads to:

- Success. All properties are valid in the model.
- Modelling error. It occurs when the model does not reflect the original system. The model has to be improved and the verification restarted.
- Design error. It is discovered that the requirements put on the system do not hold in its actual design. Before the verification can be repeated, the design together with the model have to be improved.
- Property error. It appears when the property does not reflect the specification of the system.

It is very beneficial to use model checking at the early stages of the development lifecycle to reveal inconsistency, incompleteness, and other problems in the specification and design. Nevertheless, sometimes the details of systems are not known until the implementation is started and often many errors are introduced during the implementation, despite

[2] The states themselves are not usually stored in a hash table. Instead, the numbers represeting states are stored using bitstate hashing.

the fact they are not present in the design. Furhermore, it may take a significant amount to time to create a model of a system.

Program model checking [52, 45, 51, 65] is an approach where the actual implementation, not the model, of a system is the target of the verification. Over the traditional model checking, this approach has a couple of advantages for the end users. Systems written in general programming languages (e.g., C or Java) can be analyzed by program model checkers without a high degree of expertise in model checking. Moreover, the consistency between the implementation and the explicit model does not have to be kept.

2.2.1 Java PathFinder

Java PathFinder (JPF) [11, 45] is an explicit-state program model checker for the verification of Java programs. The word *explicit* means that each program state reachable from the initial state is explored and stored for the efficient traversal of the state space. The word *program* is used to emphasise that JPF does not require any model of a system under verification – JPF simply accepts programs in their bytecode form.

Originally, JPF was a translator from Java to the Promela modelling language and it employed an automata-based model checker called Spin [44, 17]. Later, JPF has been redesigned, open-sourced, and now it consists of:[3]

- Custom-made virtual machine that executes Java programs in a different manner than a normal virtual machine. All thread interleavings and other non-deterministic characteristics of programs (e.g., random values) are considered to explore the complete state space.
- Set of properties that are checked while JPF traverses the state space. JPF comes with a plenty of generic properties used for checking that programs being verified do not contain deadlocks, assertion violations, uncaught exceptions, race conditions, etc. Additional properties respecting specific features of programs can be implemented by users.
- Set of interfaces for extending JPF (e.g., strategies for searching the state space of programs, listeners monitoring state space traversal, etc.).
- Model Java Interface (MJI) which is a bridge between the JPF virtual machine and the host virtual machine[4] . MJI is one of the JPF key features, it allows to
  - intercept calls to native Java methods and model their functionality (e.g., threading has an alternative implementation in JPF),
  - execute parts of programs in the host VM instead of JPF VM, such code is executed atomically and faster (can be used both for Java standard library classes and user classes),

[3] As most of these features are used in UnitCheck, their technical details will be described later in the text.

[4] JPF itself is written in Java and runs inside the host virtual machine.

– use the special JPF API in the programs under verification (e.g., `Verify.getInt(1,10)` instruments JPF to generate integers from 1 to 10 and systematically execute the code with each value).

To simulate non-determinism, JPF generates and goes through all non-deterministic choices. It includes various thread scheduling sequences and random values. All choices are generated by so called *choice generators*. The state space traversal is based on two concepts – backtracking and state matching. *Backtracking* is used to return into the states that have some unexplored choices left. When JPF reaches a new state, it first checks whether the state has already been visited (by the mechanism called *state matching*). If so, JPF skips the state and backtracks, if not, the snapshot of the state is stored and the model checker goes deeper into other states reachable from this new state.

The JPF operation is illustrated on a short program from Listing 2.2 which uses random values as a source of non-determinism[5].

Listing 2.2: Sample Program with Random Values

```
public class Program {
  public static void main(String[] args) {
    Random r = new Random();
    int a = r.nextInt(2);                    // (1)
    int b = 1 - r.nextInt(2);                // (2)
    int c = a / b;
    System.out.println(String.format("%d / %d = %d", a, b, c));
  }
}
```

The following steps are performed when the program is executed under JPF. For better understanding, the traversal of the program state space is depicted in Figure 2.1.

1. The interpretation of the program bytecode is started in the initial state.

2. When the line marked with (1) is reached, a new choice generator (`IntChoiceGenerator` with the set of values {0,1}) is created, the snapshot of the current state is stored, and the generator is assigned to the state. The first choice, the integer 0, is then assigned to the variable `a` in a new transition.

3. The same sequence happens on the line (2), which causes the integer 1 to be assigned in `b`.

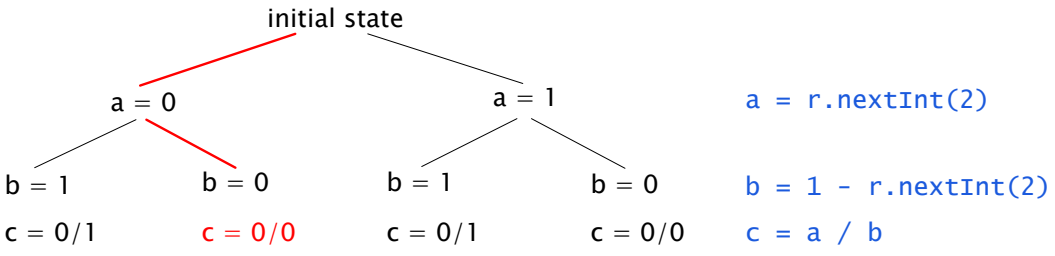
4. The result of the division is printed and because the end of the program has been reached, JPF backtracks to the previous state. There is one more choice left on the line (2) which results in 0 assigned in `b`.

5. The division by zero now causes `ArithmeticException`. Therefore, JPF reports the complete execution path leading to the error, along with all choices performed on this path (denoted by red lines in the figure).

[5] JPF has to be executed with the property `cg.enumerate_random=true` to generate all random values.

6. By default, JPF stops checking when the first error occurs. If it was configured not to stop, the figure shows how the traversal would continue.

Figure 2.1 State Space Traversal of the Sample Program



Up to now, a number of terms were used without an accurate definition. As all of them will be used henceforth in the next chapters, a short summary follows.

State
: State represents the current status of a program under verification. It holds information about the state of all threads, the state of the program memory, and the execution history (path, trace) leading to the state. Each state keeps a choice generator that produces transitions going to other states. In Figure 2.1, states are denoted by the nodes.

Transition
: Transition is a sequence of instructions that lead from one state to the next. All instructions are executed by the same thread. Every transition ends in the instruction that produces a new choice generator. In the figure, transitions are illustrated by the edges.

Choice, Choice generator
: Choice is the beginning of a new transition. It can be a different thread selected for execution or a different value from a set of random values. Choices are produced by a choice generator one by one every time the model checker backtracks to the state that has the generator assigned. In JPF, the mechanism of choice generators is very general and the set of choice types is not fixed. New choice generators can be added by users (e.g., URIs are choices in web applications).

The biggest limitation of most of existing model checkers is scalability. Nowadays, JPF can be used on source codes of about tens of KLOC. To face the scalability issues linked with the program size and state explosion, JPF uses a couple of techniques:

- Search strategy can be chosen or implemented with the respect to the character of a program (DFS, heuristics, ...). The goal is to reach an error state before JPF runs out of memory.
- Partial order reduction reduces the number of analyzed interleavings when a concurrent code is checked. It is performed on-the-fly for each thread – consecutive instructions that do not have effects outside the thread are grouped in a single transition.

- State-compression and state-abstraction techniques are applied to save a huge amount of memory during the analysis.
- MJI enables to execute the code, which users do not want to model check (e.g., library dependencies), in the host VM. The execution is faster and can save significant portions of the state space.

Apart from the state explosion problem, JPF suffers from another substantial drawback. It uses special versions of some standard Java packages, but not all of them are implemented. As an example, there is no support for `java.awt` and `java.net`. Therefore, the programs using these packages cannot be analyzed by JPF. The good point is that the number of supported native methods is rapidly increasing.

2.3 Unit Checking

This section focuses on possible combinations of the two techniques described above – unit testing and model checking. In research and industry, the term *unit checking*[6] is not widely adopted yet. First, it was proposed in [38] for symbolic model checking approach that allows verifying a unit of code, e.g., a single procedure or a collection of procedures that interact together. Other approaches and works do not explicitly mention unit checking, still they somehow mix model checking with unit testing.

The role of model checking in software testing is described in [24, 28]. Model-driven unit testing [41] generates test cases from models and derives assertions from visual contracts specified in models. In [58], authors come with an idea of splitting programs into smaller units, thus facing the state explosion. Environments for checking these units are automatically derived from specifications written by users. In [60], authors present results in generating test cases and employing model checking in the automotive industry. An approach to generate test cases to cover both the specification model and its complement is proposed in [39]. Symbolic execution is used to generate unit tests by the Symstra framework [67] and JPF [66]. In [25], mutation analysis[7] is used to generate test cases from specifications.

Most of the previous approaches generate test cases using model checking techniques. In this work, another way of combining unit testing and program model checking is proposed along with a prototype implementation. In one sentence, our solution concentrates on executing unit tests under a model checker. Before the details of the solution are described, let us explain the motivation behind the idea.

Unit testing is for years well accepted in the industry to find errors in programs, thus gaining better assurance of quality. Developers are familiar with creation of test cases and unit testing is supported in most programming languages and development environments. We believe that model checking has a great potential to become a part of the development process. It allows to find subtle errors that are hardly to be revealed by traditional testing techniques. Nowadays, a lot of model checkers for various languages exist, but still they

[6] The term *compositional verification* is also used when only units of code are analyzed. Anyway, verification is a superset of model checking because it includes other techniques, too.

[7] Mutation analysis uses mutation operators to slightly modify the program's source code.

require some sort of expertise in model checking itself or it takes a significant time for developers to start using model checking tools efficiently.

Therefore, we decided to gently move model checking closer to developers with a point of contact put on unit tests. Unit testing frameworks allow to run unit tests in a comfortable way, either from command-line, IDEs, or other development supporting tools. Our idea of unit checking is to interchange a unit testing framework with a model checker and provide users with the same (or at least very similar) convenient way of running tests. Developers do not have to change their habits and they benefit from model checking technology. Moreover, applying model checking to smaller code units also helps avoiding the state explosion problem, the main issue of the model checking tools.

The following list sums up the benefits that unit checking brings to developers over the traditional unit testing:

- ability to evaluate all thread interleavings, not just a single one,
- possibility to execute the code with all 'random' values which turns out in better program coverage,
- efficient handling of the already visited states (do not have to be revisited),
- availability of complete execution paths leading to errors found during checking.

2.3.1 UnitCheck

The previous section presented the general idea of unit checking. Here, the UnitCheck tool, a concrete implementation of the unit checking approach, is briefly introduced. Implementation details are covered in Chapter 3 and the user manual is included in Appendix A.

UnitCheck integrates two third party products. The unit testing part of the integrated work is represented by JUnit, the model checking capability is provided by Java PathFinder. These tools were chosen for UnitCheck because they

- target the same programming language – Java,
- are well-known in their areas and used in industry,
- provide rich extension interfaces which is demonstrated by a number of extensions that exist for both,
- are open source, actively developed, and have a developer community around them.

The UnitCheck tool allows for execution and evaluation of small units of code using the Java PathFinder model checker. UnitCheck accepts standard JUnit tests (in versions 3, 4) and exhaustively explores the reachable state space including all admissible thread interleavings. Moreover, the tests might feature non-deterministic choices, in which case all possible outcomes are examined. To provide users with a convenient way for using the tool, the Eclipse plugin and Ant task were implemented.

Examples of two JUnit tests that would benefit from the analysis performed by UnitCheck (in contrast to standard unit testing) are presented in Listing 2.4. The code under test, in Listing 2.3, comprises of bank accounts and bankers that sequentially deposit money to an account. The first test creates two bankers for the same account, executes them in parallel, and checks the total balance when they are finished. UnitCheck reports a test failure because the line marked with (1) is not synchronized. Therefore, with a certain thread interleaving of the two bankers, the total balance will not be correct due to the race condition. In most cases, JUnit misses this bug because it uses only one thread interleaving.

The second test demonstrates the use of a choice generator (the `Verify` class) to conveniently specify the range of values to be used in the test. UnitCheck exhaustively examines all values in the range and discovers an error, i.e., the negative account balance. When using standard unit testing, the test designer could either use a pseudorandom number generator (the `Random` class) and take a risk of missing an error, or explicitly loop through all possible values, thus obfuscating the testing code.

Listing 2.3: Incorrect Implementation of Bank System

```
public class Account {
   private double balance = 0;

   public void deposit(double a) {
      balance = balance + a; //(1)
   }
   public void withdraw(double a) {
      balance = balance - a;
   }
   public double getBalance() {
      return balance;
   }
}

public class Banker implements Runnable {
   private Account account;
   private double amount;
   private int cnt;

   @Override
   public void run() {
      for (int i=0; i < cnt; ++i) {
         account.deposit(amount);
      }
   }
}
```

Listing 2.4: Tests of the Bank System

```
@Test
public void testDepositInThreads() {
  Account account = new Account();
  Thread t1 = new Thread(new Banker(account, 5, 5));
  Thread t2 = new Thread(new Banker(account, 10, 5));

  t1.start(); t2.start();
  t1.join(); t2.join();
  assertEquals(account.getBalance(), 25 + 50, 0);
}

@Test
public void testDepositWithdraw() {
  Account account = new Account();
  int income = Verify.getInt(0, 10);
  int outcome = Verify.getInt(0, 10);

  account.deposit(income);
  assertEquals(account.getBalance(), income, 0);
  account.withdraw(outcome);
  assertEquals(account.getBalance(),
    Math.max(income - outcome, 0), 0);
}
```

Chapter 3

Unit Checking with JUnit and JPF

In this chapter, a focus is given on UnitCheck – a prototype implementation of the unit checking idea described in Section 2.3. The architecture of UnitCheck is covered in Section 3.1. It serves as a preview of the UnitCheck design without problems being deeply discussed. The following sections describe the individual parts of UnitCheck in more detail. Sources of problems and different approaches to overcome them are also presented. Section 3.2 describes how JUnit tests can be run under JPF and how to embed JPF in UnitCheck. Section 3.3 discusses the reasons for the usage of a custom runner executing JUnit tests. Section 3.4 describes how JUnit and JPF listeners are used to collect the reports about the test execution progress. The problems caused by running different parts of UnitCheck in different virtual machines are presented in Section 3.5.

3.1 UnitCheck Architecture

An overview of the UnitCheck architecture is depicted in Figure 3.1. The core module of UnitCheck, the actual integration of JPF and JUnit, is enclosed in the central box. It is compiled into a Java library so that it can be easily embedded into other Java applications (e.g., into an IDE). As an input, the core module takes an application to be analyzed, the JUnit-compliant test cases, and optionally additional properties to fine-tune JPF. The analysis is then driven and monitored via the `UnitCheckListener` interface.

It is important to note that neither JPF nor JUnit functionality and structures are directly exposed outside the core. The `UnitCheckListener` interface hides all JPF and JUnit specific details. This solution brings a couple of advantages:

- extensions (e.g., the Eclipse plugin) implement only the single (and simple) listener interface,
- in future, both JPF and JUnit can be replaced with similar tools without modifying existing extensions.

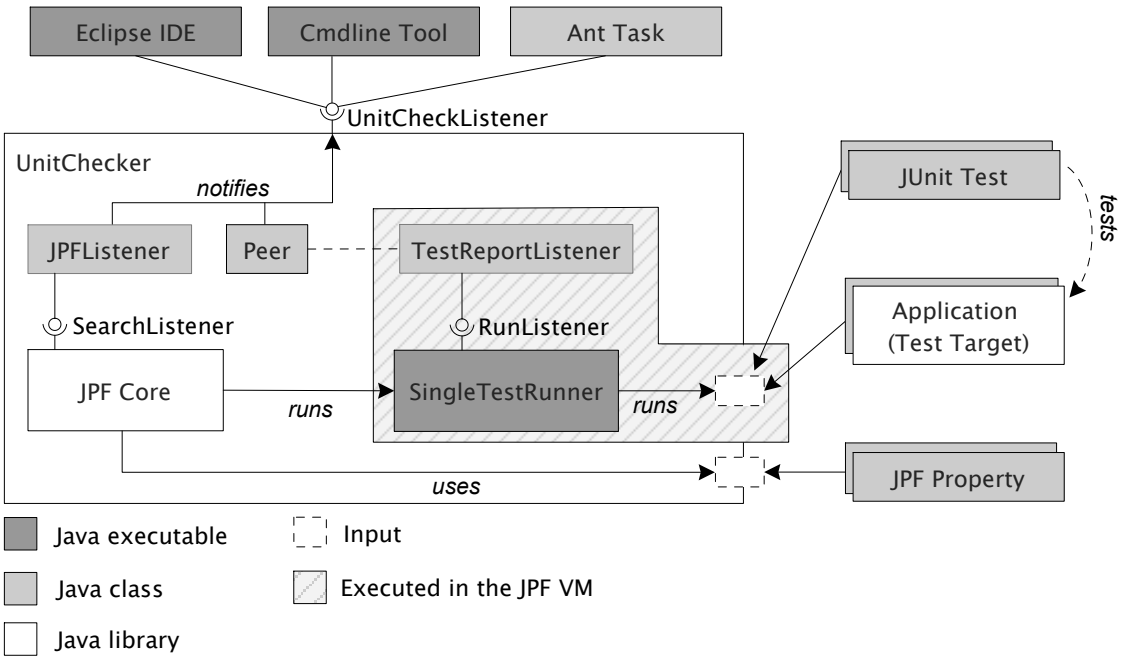
Inside the core, `UnitCheckListener` is built upon two interfaces[1] – JPF `Search-`

[1] To be accurate, the JPF `VMListener` is also implemented, but not for a substantial feature. In UnitCheck, `VMListener` is used to terminate the execution of JPF.

`Listener` and JUnit `RunListener`. `SearchListener` notifies about property violations (e.g., deadlocks) and provides complete execution histories leading to the violations. `RunListener` informs about assertion violations and other uncaught exceptions. UnitCheck processes reports from both listeners and provides them in a unified form to higher levels through `UnitCheckListener`.

When analyzing the test cases, two Java virtual machines are employed. The first one is the host virtual machine in which UnitCheck itself and the underlying JPF are executed. The second one is JPF, a special kind of virtual machine, which executes the JUnit runner. Subsequently, JUnit runs the input test cases (in Figure 3.1, the code executed inside the JPF virtual machine is explicitly marked).

Figure 3.1 UnitCheck Architecture



The information about the progress of test case execution provided by the JUnit `RunListener` interface is available only in the JPF virtual machine. To make this information accessible in the host virtual machine, the JPF Model Java Interface is used. It allows to execute parts of the application under analysis in the host virtual machine instead of the JPF virtual machine. Each class that is to be executed in the host VM has a corresponding native peer counterpart. This mechanism is used for the `TestReportListener` class.

Before we immerse into the details of UnitCheck, let us summarize the list of high level requirements put on the whole integrated work.

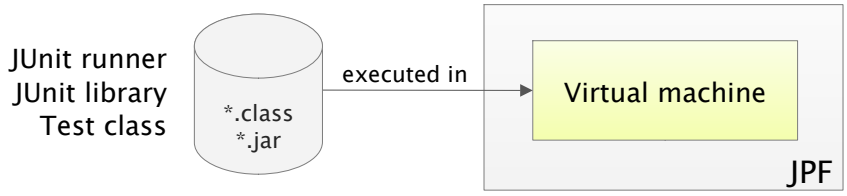
- Decoupling between the integrated tools. JPF and JUnit cannot be hard-wired together and only the public documented extension interfaces of these tools can be used. It allows for easy migrating on future versions of the integrated tools.
- Extensibility of the work. In particular, the ability to easily create various user interfaces to UnitCheck.

3.2 Running JUnit Tests Under JPF

JUnit tests are usually executed by developers in IDE plugins, Ant tasks, or using the console JUnit runner. The first two options for running JUnit are very convenient and display the test execution results and summary reports in a way suitable for reading and locating bugs. The latter option, on the other hand, is suitable for integrating JUnit in custom build scripts or other development supporting tools.

JPF can be viewed as an alternative Java virtual machine that runs Java executable programs, but in a different manner than a normal virtual machine. The JUnit console runner (`JUnitCore`) is an example of a Java executable program. The only argument of the runner describes the fully qualified name of a test class that will be executed under JUnit. It brings us to the general idea of interconnecting the JUnit and JPF tools – take a JUnit runner, a test class and run them under JPF same as any other Java application (Figure 3.2 illustrates it clearly).

Figure 3.2 Running JUnit Tests Under JPF



JPF is bundled with executables that provide a similar user interface as normal virtual machines. When running Java programs, the name of a main class along with a classpath has to be specified. In the case of running JUnit, the classpath to the JUnit library and a test which is going to be run under the `JUnitCore` runner have to be specified. Listing 3.1 shows commands used for running the `Simple` test under both virtual machines.

Listing 3.1: Running the `JUnitCore` Program

```
$ java -cp lib/junit-4.5.jar:classes org.junit.runner.JUnitCore
    cz.kebrt.unitcheck.test.Simple

$ jpf +vm.classpath=lib/junit-4.5.jar:classes
    org.junit.runner.JUnitCore cz.kebrt.unitcheck.test.Simple
```

The `jpf` script, which acts as a replacement for the normal `java` program, is suitable for the end JPF users. Besides the script, JPF provides means for embedding the model checker in custom tools which is the case of UnitCheck.

`UnitChecker` is the topmost class of the UnitCheck API used by various user interfaces (see Chapter 4). Listing 3.2 shows how JPF is embedded in `UnitChecker`.

1. JPF configuration is created and initialized. It accepts the same command-line arguments as the `jpf` script (classpath, name of the JUnit runner class, etc.). The configuration can be used to inject custom objects to be further accessible (e.g., in MJI peers).

2. Same as `UnitChecker` is the main class of UnitCheck, `JPF` is the main class of Java PathFinder. The listeners added to `JPF`, which monitor the state space traversal, will be described later. After invoking the `run` method, `JPF` starts to execute JUnit (the runner class passed as an argument to the configuration) same as the `jpf` script.

Listing 3.2: JPF Embedded in UnitCheck

```
Config jpfConfig = JPF.createConfig(cmdLine);
jpfConfig.put(JPFJUNIT_LISTENER_KEY, jpfjunitListener);

JPF jpf = new JPF(jpfConfig);
jpf.addListener(jpfjunitListener);
jpf.run();
```

NOTE

The first attempt to run a JUnit test under JPF was not successful. As was noted in Section 2.2.1, JPF uses its own versions of the standard Java libraries and unfortunately, not all the functionality is implemented. Initially, there were missing implementations for a couple methods from the `java.io` and `java.lang.reflect` packages, which are used by the JUnit tool (e.g., `Class.getConstructors()`).

The missing methods were implemented in a cooperation with the JPF team. As a result, the JUnit runner was successfully executed under the JPF model checker.

3.3 More Tests in One JPF Run

The introduction to JPF (Section 2.2.1) described how the complete state space of input programs is explored. On certain program places (called transition or choice points), JPF creates choice generators and uses them to successively traverse more execution branches starting at these points. After one branch has been explored and JPF backtracked to the transition point, the next choice is selected from the generator and the corresponding branch is traversed. When JPF reaches a program state that has already been visited, it does not go through the state again. For further discussion, it is important to note the state equality is checked only in transition points.

Two scenarios will be described when the backtracking mechanism is applied on JUnit tests executed under JPF. In a simple setup comprising of a class with only one test method, the JPF backtracking mechanism may cause the test to finish multiple times during checking (e.g., when the test uses more threads and JPF analyzes all thread interleavings). From the UnitCheck point of view, it is not very difficult to handle and report multiple finish of a single test as it will be described later. A problem arises when a test class containing multiple tests is checked. Listing 3.3 contains a test class with two test

methods. The first one, the `addition` method, forces JPF to successively generate two integer values into the `num` variable using the JPF `Verify` class and to execute the rest of the test with each value. At this choice point, JPF creates two execution branches – the first continues to execute with the value 1 in the `num` variable, the second will use the value 2.

Listing 3.3: Test Class Using the `Verify` Choice Generator

```
@Test
public void addition() {
  int num = Verify.getInt(1,2); // choice point
  assertTrue(SimpleMath.isPositive(num));
}
@Test
public void division() {
  ...
}
```

Listing 3.4 shows how the JUnit runner (the `JUnitCore` class mentioned in the previous section) executes an input test class. It runs all tests one by one – when a test method call is finished, the next one is invoked.

Listing 3.4: `JUnitCore` Runner

```
// JUnit executes tests one by one
public class JUnitCore {
  public static void main(String args[]) {
    TestClass testClass = getTestClass(args);
    foreach (Test test : testClass) {
      test.invoke();
    }
  }
}
```

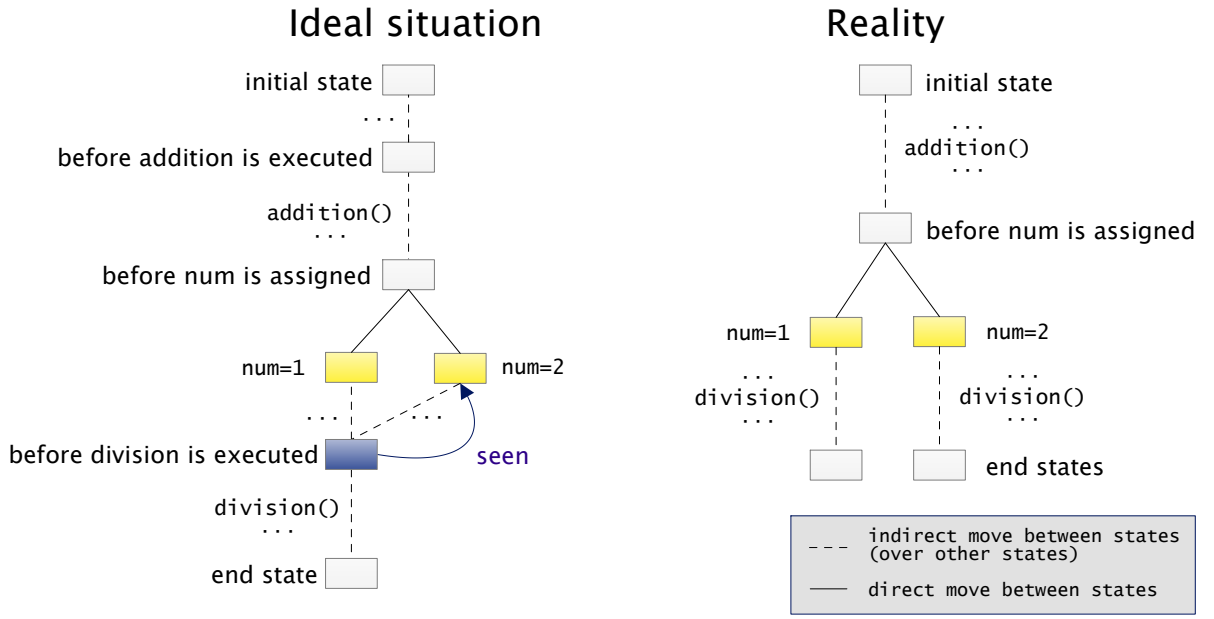
When this JUnit runner is used to execute tests from Listing 3.3, the following will happen.

1. The `addition` test with the value 1 in the `num` variable is finished.
2. The `division` test is executed.
3. JPF backtracks to the `addition` method and assigns the value 2 into the `num` variable.
4. The `addition` test ends for the second time. Unfortunately, this is not everything, also the `division` test is executed for the second time. This the problem which we need to avoid because both tests are independent and thus, the `division` test does not have to be invoked twice.

Let us have a look at the situation from the JPF perspective. The right part of Figure 3.3 illustrates what actually happens – JPF executes a part of the program in two branches and in each of them both test methods are invoked. The reason for this is the fact there is no potential transition point between the first test is finished and the second

test started (transition points do not necessarily include method invocations). Therefore, before the `division` method is executed, the equality with previously visited states is not checked.

Figure 3.3 More Tests in One JPF Run



To make it clearer, the left part of the figure describes what would happen if there was a transition point before test method invocations. The `division` test would be executed in a new transition, starting in the blue state. When the test was about to run for the second time, JPF would detect that it went through the blue state before and would not invoke the `division` test for the second time. It is really true that the execution state before the `division` method is invoked is always the same. The `num` variable, as a local variable from the `addition` method, does not have any impact on the rest of test methods.

3.3.1 Custom JUnit Runner

Two solutions of this problem were considered. Both have their advantages and disadvantages.

- Extend JPF with a custom choice generator that creates a transition point before each test method is invoked. This would result in an ideal situation described in Figure 3.3. The implementation of the generator requires to check every method invocation in a program under analysis and to find out whether it is a JUnit test method or just a common Java method. The rules for detecting a JUnit test method are not very straightforward and differ between JUnit 3 and 4. Therefore, the generator might notably slow down the overall unit checking process.
- Execute only one test method in one JPF run. Unfortunately, test runners bundled with the JUnit tool cannot invoke single test methods. Therefore, a custom JUnit

runner which is capable of running single JUnit tests must be implemented. Such a runner does not modify any JUnit internal code and is used as a replacement for the default `JUnitCore` which always invokes all test methods of an input test class.

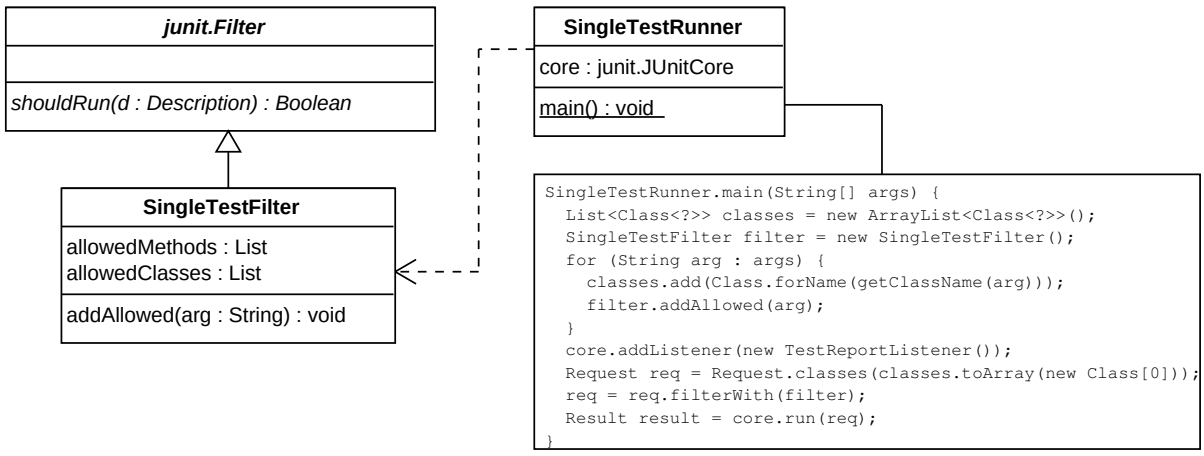
The only known disadvantage of this solution is connected with the `BeforeClass` and `AfterClass` JUnit annotations. They are used to mark methods to be invoked only once by JUnit – either before the first test method of a class is invoked or after the last test method is finished. Before JUnit invokes test methods, it creates an instance of a test class. If only one test method is checked in a JPF run, such instance is constructed more times and also the methods marked with `BeforeClass` and `AfterClass` are invoked more times (once for each test method). Semantically, it is correct, but the overall performance is negatively influenced. The impact is more apparent when the *before/after* methods are very time-consuming.

It is important to notice that the solution was now described only from the developer perspective. From the user perspective, of course, it does not mean that users must check test cases method by method. Same as when running JUnit, users can specify the whole test class to be checked . However, the underlying code runs JPF multiple times, once for each test method.

We decided for the second approach which is easier to be implemented and the problem with *before/after* is not so significant because these annotations are new in JUnit 4 and not widely used among developers.

Figure 3.4 shows how the custom JUnit runner, which is capable of running single test methods, is implemented. The arguments of the runner describe either whole test classes to be run (using fully-qualified class names), or single test methods (using the *class-name#method-name* notation)[2]. `SingleTestRunner` uses the default `JUnitCore` runner for running a filtered list of test cases. According to the arguments, `SingleTestFilter` filters out the test methods that were not explicitly specified.

Figure 3.4 JUnit Runner for Running Single Tests

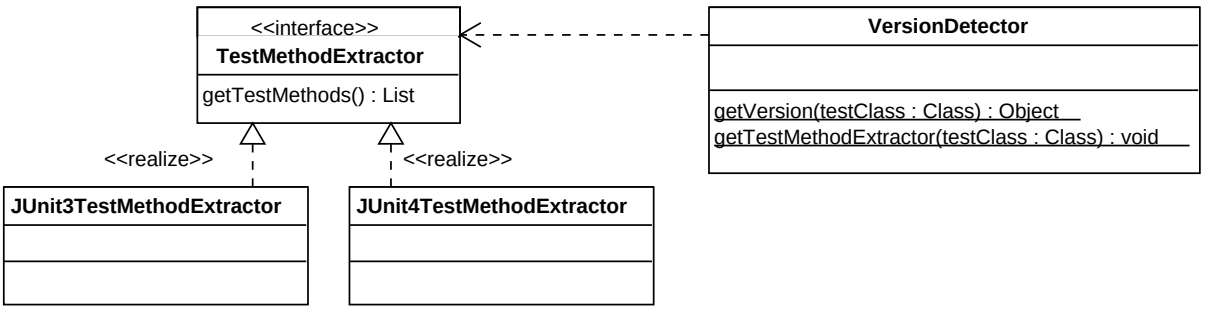


Internally, UnitCheck analyzes a single test method by executing `SingleTestRunner` under `JPF`. Externally, UnitCheck allows users to analyze whole test classes by applying

[2] In UnitCheck, only the second way of specifying arguments is used.

the previous scheme on each test method of a class. Before it can be done, a list of all test methods within the class has to be extracted. Figure 3.5 lists UnitCheck classes that are used to extract test methods' names from JUnit test classes.

Figure 3.5 Extraction of Test Method Names



3.4 Collecting Information About Test Execution

Both JUnit and JPF provide various information about their execution progress. Basically, JUnit reports test failures and successes, while JPF allows to monitor the state space traversal down to the level of individual instructions. These monitoring mechanisms can be used to gather statistics about the execution, present results to users, etc.

In UnitCheck, JUnit tests are analyzed using the JPF model checker. Each executed test can finish either successfully or with an error. In case of a success, it is usually enough to know there was no uncaught exception, property violation, or other type of failure. JUnit only reports the time it took to execute the test. On the other hand, when the test fails, it is important to provide as much information as possible to make it easy for developers to find the problem in the implementation that caused the test to fail.

JUnit and JPF report different kinds of errors. JUnit informs about assertion violations and other uncaught exceptions, whereas JPF notifies about property violations. JPF generally provides more information about test failures in comparison with JUnit. Java PathFinder reports the complete execution history leading to properties violated during the test execution and also the state of all threads at the moment of a test failure (*thread snapshot*). Therefore, the goal was to offer developers this additional information, especially the execution history, even for errors reported by JUnit[3].

3.4.1 JUnit Listener

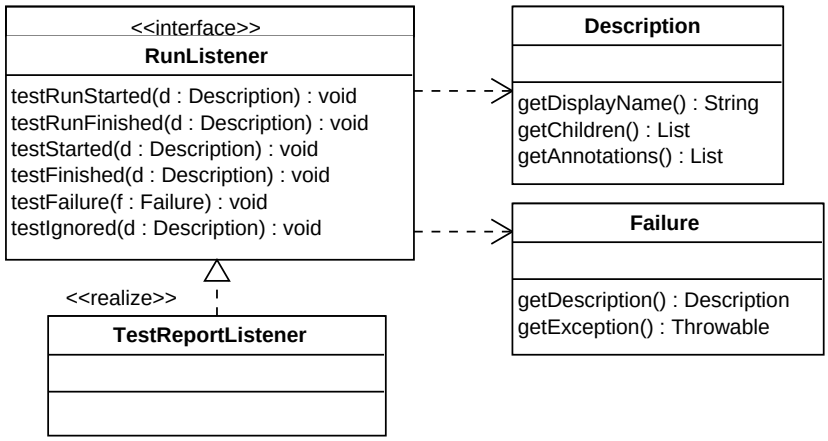
To gather the information about test execution, JUnit and JPF listeners are used. JUnit provides only one listener interface called `RunListener` that notifies when a test was started and finished and whether an assertion was violated or some other exception oc-

[3] All exceptions thrown by tests are caught in a JUnit test evaluator and thus, the JPF model checker does not know that any problem occurred.

curred. Figure 3.6 describes the listener together with all classes it uses[4]:

- `RunListener` provides a series of methods called when the execution of a test class (methods prefixed with `testRun`) or a single test (methods prefixed with only `run`) is started and finished. The `testFailure` method is called when a test fails (e.g., when an unexpected exception is thrown by the test).
- `Description` describes a test class, test suite, or a single test. The `getDisplayName` method returns the textual characterization of an element described by an instance of `Description`. In case of a class or suite, descriptions of child elements can be retrieved, too.
- `Failure` holds the exception that caused a test failure and the description of the corresponding test.

Figure 3.6 JUnit `RunListener` and Related Classes



In UnitCheck, the implementation of `RunListener` is called `TestReportListener` and is used to collect the information about test execution progress, especially about test failures. An instance of `TestReportListener` is added to the JUnit runner as can be seen in Figure 3.4. Section 3.5 contains more information about the implementation of the listener.

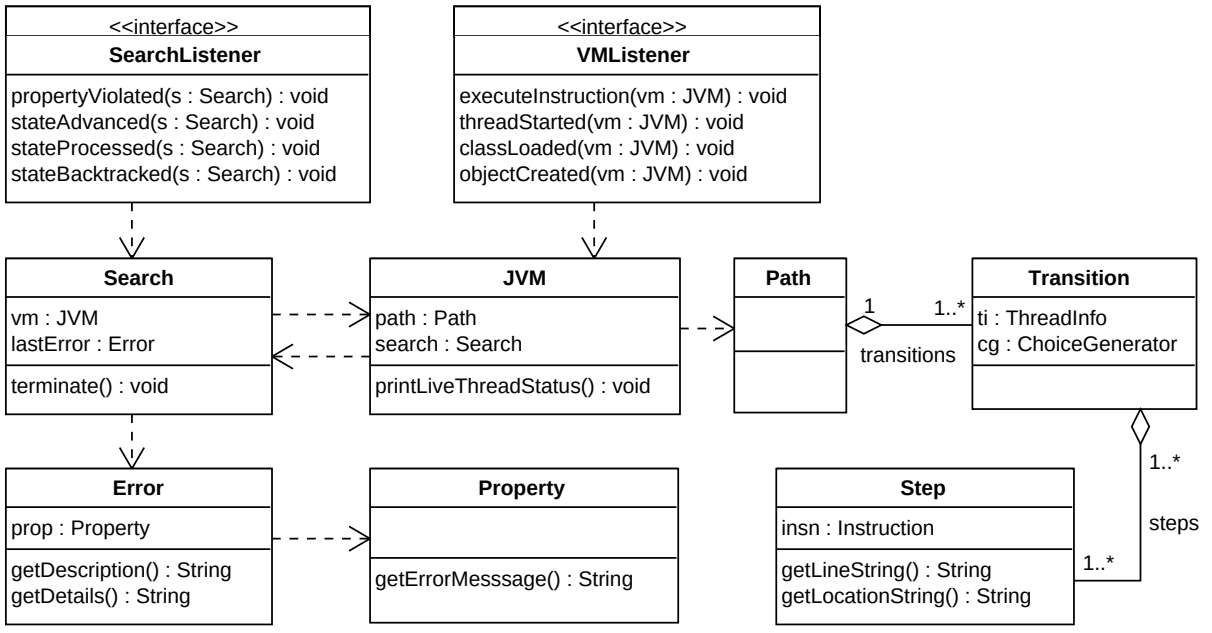
3.4.2 JPF Listeners

JPF provides two different listeners. `VMListener` monitors the VM processing (executing instructions, threading, manipulating objects, etc.). In UnitCheck, the listener is used to terminate the execution of JPF. `SearchListener` monitors the state space traversal and notifies about property violations which include deadlocks, user-defined properties, etc. The JPF listeners along with the related classes are depicted in Figure 3.7.

[4] For simplicity, the API provided by these classes which is not used in UnitCheck is not covered in the figure.

- `SearchListener` – the only callback used in the UnitCheck implementation is the `propertyViolated` method.
- `VMListener` – in UnitCheck, the `executeInstruction` method checks whether the termination of JPF was requested.
- `Search` is an abstract class which is in charge of searching the state space. UnitCheck uses it to terminate the search and to access the JPF virtual machine and eventual errors that occurred while searching. Each `Error` holds a `Property` whose violation caused the error.
- `JVM` represents the virtual machine. Besides a number of other features, it allows to print the thread snapshot and to get the execution path starting at the initial state and leading to the current state.
- `Path` holds the list of transitions. Each `Transition` keeps its choice generator and information about the thread that executed the transition. Transitions are further composed of steps which represent bytecode instructions. Each `Instruction` object may return its line and location in the source code. The Path structure, as a whole, allows to show users the execution traces leading to errors. Depending on the user interface, the traces can be printed on the console, displayed in a graphical UI, etc.

Figure 3.7 JPF Listeners and Related Classes



3.4.3 UnitCheck Listener

As was stated in Section 3.1, the UnitCheck core module is a Java library with no user interface – various UIs are built on top of the library. To allow these user interfaces (or *extensions* in general) to show the reports about the test execution progress, the

information provided by `RunListener` and `SearchListener` has to be exposed outside the library.

UnitCheck offers a similar notifying interface as the both integrated tools. To make the extensions independent on JUnit and JPF, one unified listener is built on top of both. It is called `UnitCheckListener` and it makes no difference between the errors reported by JUnit and JPF. Section 4.1 describes the listener from the perspective of extensions and how the listener is implemented in the Eclipse plugin and Ant task.

The structures provided by `RunListener` and `SeachListener` have their simplified counterparts provided by `UnitCheckListener` (e.g., `StepHolder` instead of `Step`). Besides the independece on JUnit and JPF, other reasons for these dual structures are:

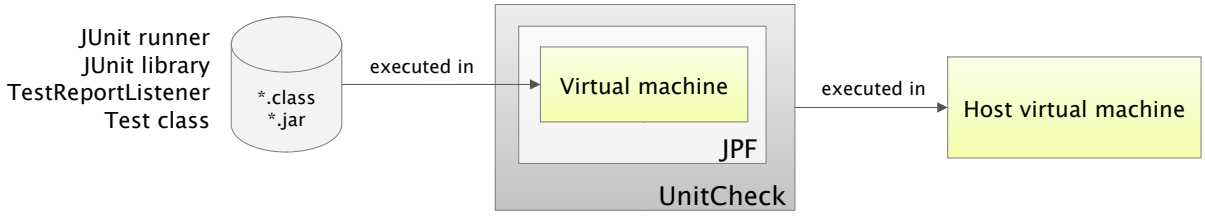
- original structures, especially in JPF, are very huge and contain a lot of internal data that should not be presented to UnitCheck users,
- complications with `RunListener` and MJI (covered in Section 3.5),
- option to filter the UnitCheck and JUnit code out of error traces[5],
- possibility to present the original structures in a different way (e.g., UnitCheck allows for the bidirectional traversal of transitions).

Instances of `UnitCheckListener` are notified in `JPFJUnitListener`. It implements both JPF listeners (`SearchListener`, `VMListener`) and custom `TestRunListener` which is a dual interface for JUnit `RunListener` (used for similar reasons as the JUnit dual structures described above). As a result, `JPFJUnitListener` receives all information from JPF and JUnit, combines it together, and provides in a unified form to user interfaces through the instances of `UnitCheckListener`.

3.5 JUnit as a Part of UnitCheck Input

In the previous section, a complication with the JUnit `RunListener` was only foreshadowed. Now the situation behind the listener will be described in more detail along with a proposed solution.

Figure 3.8 JUnit as a Part of UnitCheck Input



`TestReportListener`, implementation of the `RunListener` interface, is used as a source of reports about the execution of JUnit tests under JPF. After some processing,

[5] As the JUnit runner and the whole JUnit library are parts of the UnitCheck input, the error traces contain listings of their code.

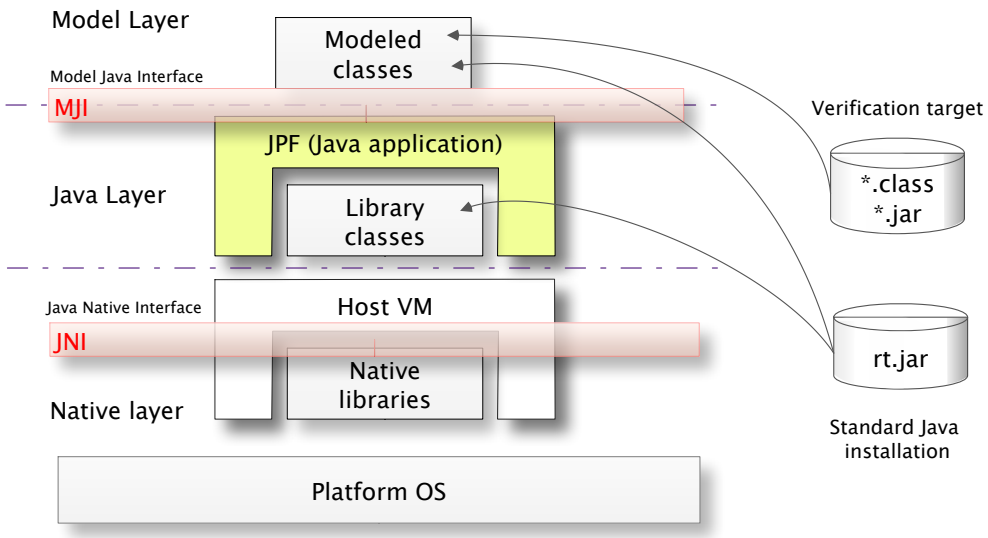
the reports together with the information provided by JPF are forwarded to all registered UnitCheck listeners (see Chapter 4). So far, everything looks good and easy, but there is a major problem with the JUnit listener – Figure 3.8 helps to better understand it.

In the figure, the UnitCheck box can represent for example the UnitCheck Eclipse plugin. It has Java PathFinder embedded in itself. Two Java virtual machines depicted in the figure are crucial. Internally, they are very unlike each other, they have different object and class models, and objects created in one VM cannot be easily accessed in the second VM. Tests to be analyzed by UnitCheck, JUnit library, and the custom JUnit runner make up the JPF input and thus, they are all executed in the JPF VM. An instance of `TestReportListener` is created by the runner within the JPF VM. On the other hand, the rest of UnitCheck (JPF listener, UI, etc.) is executed in the host VM. Therefore, `TestReportListener` is hidden for the code residing in the UnitCheck box.

3.5.1 MJI Applied on JUnit Listener

In Section 2.2.1, MJI was briefly introduced. Among other things, it allows to execute parts of applications analyzed by JPF in the host VM instead of JPF VM. When it is applied on `TestReportListener`, UnitCheck will be able to see the listener and collect the information it provides.

Figure 3.9 Model Java Interface



Model Java Interface (MJI) is very similar to Java Native Interface (JNI) as depicted in Figure 3.9[6]. JNI is used to delegate execution from the Java level (or bytecode in other words) down to the native layer which is the machine code. By analogy, MJI is used to delegate execution from the bytecode controlled by JPF down to the host virtual machine, down to the Java layer.

[6] Figure taken from http://javapathfinder.sourceforge.net.

That exactly matches the situation with `TestReportListener`. Although the listener is a part of model classes, there is a need to interact with it from the Java layer. This is the reason why the listener is implemented using MJI, which moves the execution layer of this listener from the model one down to the Java layer. As a result of this solution, it is possible in the Java layer to collect and process the results from both JUnit and JPF listeners.

The code executed in the host VM using MJI is split in two types of classes residing in different layers.

- Model classes are executed in the JPF VM. Methods that are to be invoked in the host VM are marked as *native*.
- Native peer classes implement the native methods of the corresponding model classes. They are always executed within the host VM.

Listing 3.5 shows a piece of the `TestReportListener` model class. For each overridden method of `RunListener`, there is another method marked as native that is executed by the corresponding native peer class. There are two reasons for a pair of methods:

- It is easier to access the arguments of basic data types (strings, integers, etc.) in the peer (e.g., `String` instead of `Description`).
- The MJI interface allows to access the instances of model classes using fields (calling methods is an experimental feature). Not to rely on the names of fields used in the implementation of an original class (e.g., `Throwable`), a custom dual class (e.g., `ThrowableHolder`) is created. Such dual class can be accessed by field names without a fear that a change in the field names of the original class will break the functionality.

Listing 3.5: `TestReportListener` Model Class

```
public class TestReportListener extends RunListener {
  @Override
  public void testRunStarted(Description description) {
    testRunStartedNative(description.getDisplayName());
  }
  public native void testRunStartedNative(String description);
}
```

The corresponding native peer class is located in Listing 3.6. A special name mangling scheme combining the original package and class name is used for peer classes. Implementations of native methods have to be *public static*, arguments are translated into integer handles, and two special arguments are added.

`MJIEnv` allows to access (read, write) JPF controlled objects using the handles. Some other JPF structures (e.g., configuration, virtual machine) are also accessible via `MJIEnv`. The `rObj` argument is a handle to the corresponding JPF *this* object (in this case, an instance of `TestReportListener`).

Finally, an instance of custom `TestRunListener`, injected in the JPF configuration (see Listing 3.2), is notified from the peer. The original JUnit `RunListener` cannot be

used instead because some of the JUnit structures provided by this listener (e.g. `Result`) cannot be fully reconstructed in the Java layer (public constructors are not available).

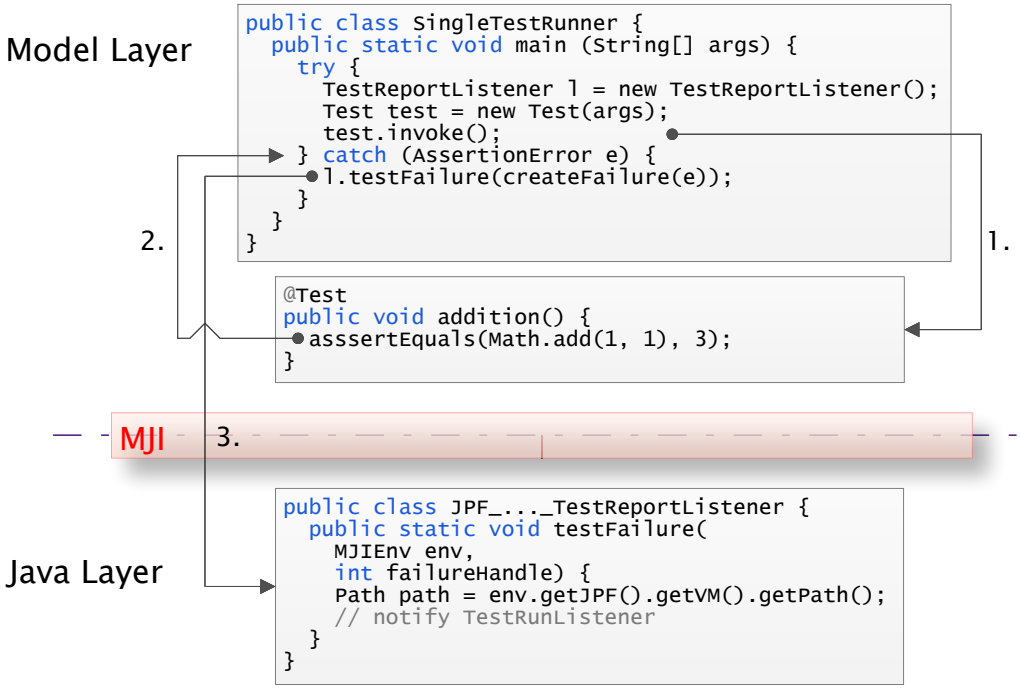
Listing 3.6: `TestReportListener` Native Peer Class

```
public class JPF_cz_kebrt_unitcheck_junit_TestReportListener {
  public static void testRunStartedNative(MJIEnv env,
      int rObj, int rDesc) {
    getListener(env).testRunStarted(
      new Description(env.getStringObject(rDesc)));
  }
  private static TestRunListener getListener(MJIEnv env) {
   return (TestRunListener) env.getConfig().get(
     UnitChecker.JPFJUNIT_LISTENER_KEY);
  }
```

3.5.2 JUnit – Perfect Exception Firewall

JUnit catches all exceptions thrown by the code under test and decides whether it was ok to throw such an exception or not[7]. No exception leaves the JUnit runner, and therefore, JPF never detects the violation of the uncaught exception property and cannot report the execution history leading to the exception. For JPF, it simply looks the program finished successfully.

Figure 3.10 Model Java Interface – Concrete Example



[7] When a test method is annotated with `@Test(expected=...)`, JUnit evaluates the test as failed only when the particular exception is not thrown.

To provide users with complete execution histories even for the exceptions reported by JUnit, Model Java Interface is used again. In the native peer of the JUnit listener, the `Path` object, describing the execution from the initial state up to the current one, can be read from the `MJIEnv` object.

The situation with the JUnit exceptions is depicted in Figure 3.10 which also serves as an example of what was presented in the previous section. A simplified version of the JUnit runner invokes a test according to the arguments passed to the runner. The assertion for integer equality does not hold in the test. The figure shows the execution layers in which the assertion exception is handled. First, an object describing the exception is created and passed into the listener. The `testFailure` method, implemented in the native peer class, receives the handles to all arguments of the original method and uses them to read the original values. We see a handle to the `Failure` object that wraps the test method description and the assertion exception. Finally, `TestRunListener` is notified about the test failure – information about the exception is a combined with the execution history.

This was not the only problem related to JUnit as a perfect exception firewall. Suppose that a test finished with an assertion error. Nevertheless, from the JPF point of view, nothing happened and if there are some unexplored states, JPF will backtrack and continue to analyze the test. However, developers usually want only the first error to be reported and to stop the checking afterwards. The `terminate` method of the JPF `Search` object stops the search of the state space[8].

An alternative, but no so elegant, way of terminating JPF is a custom JPF property that would be violated after the first JUnit exception occurs. Before JPF backtracks, it checks the state of all properties. If any is violated, JPF stops the analysis[9].

[8] The same approach for terminating JPF is used in `JPFJUnitListener` which checks whether the termination was requested by the user (e.g., by clicking the stop button in the Eclipse plugin).

[9] By default, JPF is configured to stop when a JPF property is violated (`search.multiple_errors` configuration property).

Chapter 4

UnitCheck in 3rd Party Programs

Up to now, only the implementation of the UnitCheck core was presented. To bring the unit checking capabilities to developers, a user interface for UnitCheck has to be created. Section 4.1 lists the most important concepts that allow to embed UnitCheck in a third party tool. Implementations of two user interfaces for running UnitCheck, Ant task and Eclipse plugin, are demonstrated in Section 4.2 and Section 4.3.

4.1 Introduction

This section describes what needs to be done in order to embed UnitCheck in a third party application. Figure 4.1 shows the most important parts of the UnitCheck external interface.

UnitChecker
: The topmost class of the whole tool. It is initialized with a configuration specifying a set of unit tests to be checked. Two methods for running and terminating the checking process are provided. Termination can be used for example in an IDE plugin when a user clicks a stop button.

Config
: Represents a configuration of one run of the `UnitChecker` class. One or more JUnit test classes (or even single methods) along with their classpath must be specified. JPF properties may be overridden here and a couple of other attributes can also be set (e.g., filtering of UnitCheck traces in error reports).

UnitCheckListener
: The listener provides a mechanism for reporting of what is happening in the checking process. Four callbacks must be implemented. Each of them identifies a test method and provides the results collected while checking the test method. The callbacks whose name start with `checkRun` are used when the checking of a test method is started or completely finished. The `checkFinished` callback is invoked every time a particular test method is finished.[1]

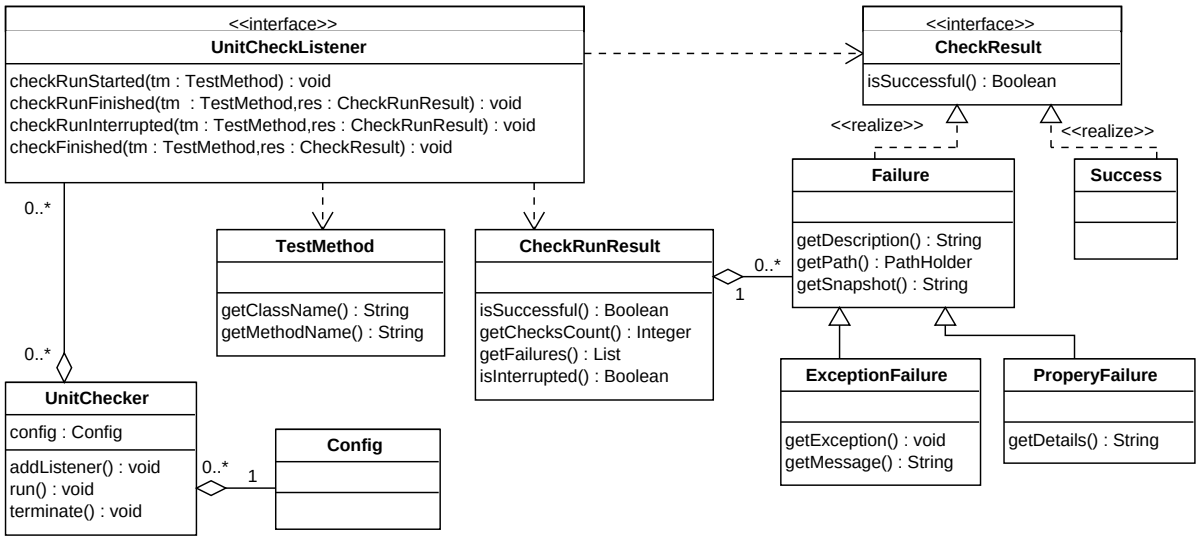
[1] Each test method may finish a number of times because JPF explores all possible execution paths.

CheckResult, Failure, Success
: The checking of a test method may finish with a failure or successfully. In case of a failure, more information is supplied to make the debugging easier. The complete execution path that led to the failure is provided. The `PathHolder` class represents quite a large but not complicated path structure and thus, it will not be covered in this text. Additionally, the snapshot of all threads at the moment of the failure is also available.

PropertyFailure, ExceptionFailure
: Two concrete types of failures exist. Those that are caused by uncaught exceptions and those that are results of violated JPF properties.

CheckRunResult
: Summarizes the results of checking one test method. The `checksCount` attribute counts how many times the method finished (in other words how many times the `checkFinished` callback was invoked for the test method). The list of failures collected during the checking is either empty (in case of overall success) or contains exactly one failure (UnitCheck is usually configured to stop checking after the first test failure is encountered).

Figure 4.1 External Interface of UnitCheck



The procedure of creating a new user interface to UnitCheck (e.g., an IDE plugin) usually consists of the following steps.

1. Create a user interface for configuring UnitCheck so that the `Config` class can be correctly filled.
2. Implement `UnitCheckListener` which displays provided results in a user friendly way.
3. Instantiate the `UnitChecker` class with a configuration and call the `run` method.

4.2 Ant Task

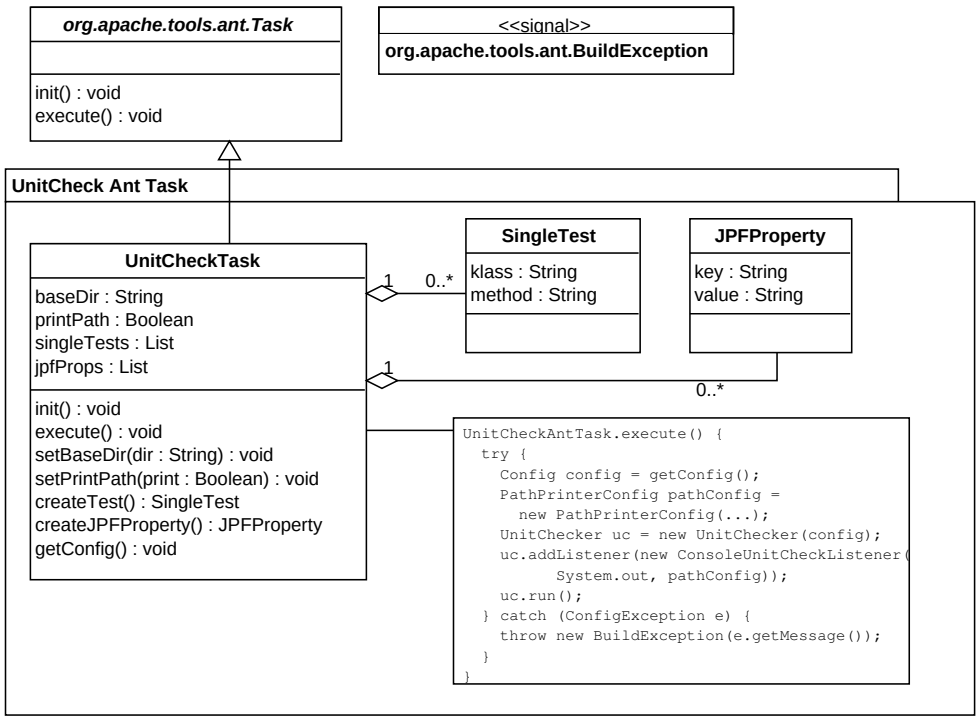
The UnitCheck task brings a support for unit checking in Ant [2], a very popular tool used in the Java software development. A short Listing 4.1 shows how to use the task in a build file. This example will help us to better understand the concepts used in the implementation.

Listing 4.1: Usage of the UnitCheck Ant Task

```
<unitcheck basedir="${unitcheck.task.dir}">
  <test class="daisy.unittest.CreatFileTest"
        method="testCreatLongFilename" />
</unitcheck>
```

The most important parts of the task are depicted in Figure 4.2. The `UnitCheckTask` class represents the core of the task. As well as other Ant tasks, it extends the `Task` base class and overrides the methods for initialization and execution. For each task's attribute, a setter method, whose name begins with `set` followed by attribute name, must be provided. For each nested element, a class representing the element (with the same rules for attributes as the root element) must be implemented. Also the `create` method returning an instance of such a class must be implemented in the parent class. In Figure 4.2, it is the case of the `SingleTest` (`test` element) and `JPFProperty` (`jpfproperty` element) classes.

Figure 4.2 Structure of the UnitCheck Ant Task



When the `unitcheck` element appears in an Ant target that is executed, Ant performs the following steps.

1. An instance of the `UnitCheckTask` class is created and the `init` method is called.
2. For each nested element, the corresponding `create` method is invoked.
3. All attributes of this task get set via their corresponding `set` methods.
4. All attributes of all nested elements get set via their corresponding `set` methods.
5. The `execute` method is called.

The typical scenario described in the introduction of this chapter is then performed in the `execute` method. A configuration is created from the data collected the in previous steps. A listener[2] that prints results on the standard output is then passed to the `UnitChecker` instance and finally the `run` method is executed.

4.3 Eclipse Plugin

The UnitCheck plugin allows for unit checking within the Eclipse IDE. The plugin uses and extends a lot of standard Eclipse solutions to make its usage intuitive and convenient. It includes Eclipse views, launch configurations, launch shortcuts, jobs, a preference page, and other Eclipse concepts thoroughly described in [33].

4.3.1 Structure of the Plugin

Since Eclipse runs on Equinox [9], which is an OSGi container [15], there have always been arguments whether the word *plugin* should be used for the same thing as the word *bundle* from the OSGi specifications (bundle is a package of code that runs under an OSGi container). In this work, the word plugin is used because it is more widespread among the Eclipse community.

Each plugin has a form of a JAR file containing

- code of the plugin (class files),
- resources (icons, help, localization),
- `MANIFEST.MF` (plugin manifest)
- `plugin.xml` (describes extension points).

The plugin is described in a manifest file (Listing 4.2 shows a part of the UnitCheck manifest) which is used by an OSGi run-time. A separate classloader is created for each plugin and thus, the dependencies on other plugins and JAR files must be specified in the manifest.

[2] `ConsoleUnitCheckListener` implements the `UnitCheckListener` interface and used also in the command-line application and Eclipse plugin.

Listing 4.2: UnitCheck `MANIFEST.MF` File

```
Manifest-Version: 1.0
Bundle-Name: Unitcheck Eclipse Plug-in
Bundle-Version: 1.0.12
Bundle-Activator: cz.kebrt.unitcheck.eclipse.UnitCheckPlugin
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
 org.eclipse.jdt.core;bundle-version="3.4.2",
 org.eclipse.core.resources;bundle-version="3.4.1",
 org.eclipse.jdt.launching;bundle-version="3.4.1"
Bundle-ClassPath: .,
 jars/jpf.jar,
 jars/junit-4.5.jar
```

Prior to version Eclipse 3.0, the contents of `MANIFEST.MF` files was included in `plugin.xml` files. After migrating to the OSGi architecture, the only Eclipse-specific properties left are *extension points* and *extensions*. The idea of extensions is very general and is used for a variety of features in Eclipse. A plugin describes an extension point, an abstract implementation of some general feature, using the `extension-point` element in its `plugin.xml` file. The extension point specifies what every concrete implementation (extension) needs to provide and implement.

Eclipse views are a nice example of an extension point. Listing 4.3 contains a piece of the UnitCheck `plugin.xml` file describing the plugin's views. As specified by the `org.eclipse.ui` plugin, each view must provide a window icon, name and a class implementing the `org.eclipse.ui.IViewPart` interface.

Listing 4.3: UnitCheck `plugin.xml` File

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension point="org.eclipse.ui.views">
      <category id="cz.kebrt.unitcheck.eclipse"
                name="%view.category.title" />
      <view category="cz.kebrt.unitcheck.eclipse"
            class="cz.kebrt.unitcheck.eclipse.tree.ResultTreeView"
            icon="icons/check.gif"
            id="cz.kebrt.unitcheck.eclipse.views.ResultTreeView"
            name="%view.results.title" />
   </extension>
</plugin>
```

In Listing 4.2, the `UnitCheckPlugin` class is given as an activator. During the startup, the activator is the first plugin's class instantiated by Eclipse. It allows to keep the plugin's lifecycle under control using the `start` and `stop` methods. This is not the case of the UnitCheck plugin, which does not hold any resources like database connections or files which need to be closed at exit. Therefore, this class is used only for the initialization of default values in the preferences.

4.3.2 Running the Checking Process

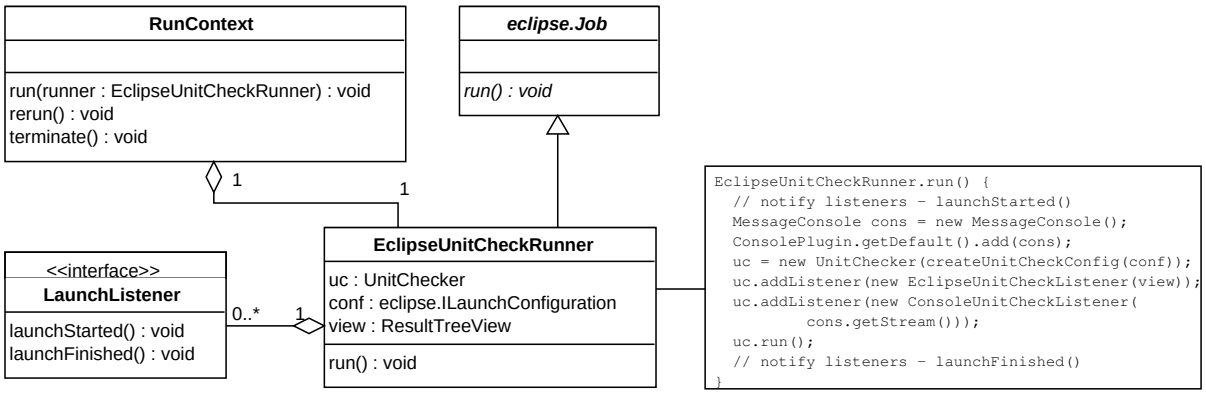
The steps necessary to incorporate UnitCheck in other applications were described in Section 4.1. In case of the Eclipse plugin, the procedure is more complicated than in the case of the Ant task but the idea remains same. Figure 4.3 shows where the `UnitChecker` class, the main entry point to the UnitCheck core, is instantiated. In the centre of the figure, there is the `UnitCheckEclipseRunner` class which runs the `UnitChecker` instance. The runner extends the Eclipse `Job` class which allows to display the progress of execution in the Progress view. The runner is initialized with

- an Eclipse configuration either coming from a launch configuration dialog or automatically created by a launch shortcut,
- a result view used for displaying the progress of checking,
- a set of listeners that are notified when the checking is started and finished.

The checking is started by `RunContext` which remembers the instance of `UnitCheck-EclipseRunner` to be able to stop or rerun the latest checking process. Two UnitCheck listeners are created and passed to `UnitChecker`:

- `ConsoleUnitCheckListener` uses the stream provided by the Console plugin to print the results in the Eclipse console.
- `EclipseUnitCheckListener` provides the result view with the output of checking.

Figure 4.3 Running UnitCheck Within the Plugin

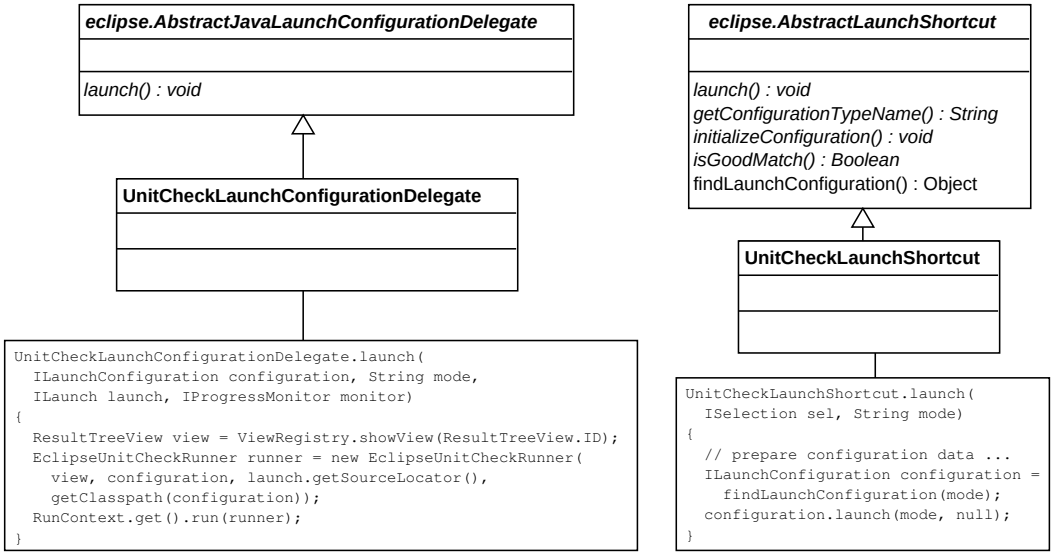


So far, the way of using `UnitChecker` was described from the internal point of view. Eclipse provides *launch configurations* and *launch shortcuts* to allow users to run external programs within the IDE. The configurations can be edited in dialogs, saved, and reused later. The shortcuts allow to run external programs quickly, without a chance to modify default settings, just by selecting a resource and an item in the run menu. In order to use both of these concepts, the following extension points must be implemented in a plugin.

- `org.eclipse.debug.core.launchConfigurationTypes` – defines a new type of launch configuration and a delegate (implementation of `ILaunchConfiguration-Delegate`) that is used for running these configurations.
- `org.eclipse.debug.ui.launchConfigurationTypeImages` – assigns an icon to a configuration type.
- `org.eclipse.debug.ui.launchConfigurationTabGroups` – assigns a graphic dialog (implementation of `ILaunchConfigurationTabGroup`) to a configuration type.
- `org.eclipse.debug.ui.launchShortcuts` – assigns a shortcut (implementation of `ILaunchShortcut`) to a configuration type and defines rules for displaying an item in the run menu.

Figure 4.4 shows concrete implementations of some of the extension points mentioned above. `UnitCheckLaunchConfigurationDelegate` extends the abstract class, which is useful for delegates that run Java programs (which is the case of UnitCheck and JPF) because it allows for easy manipulation with classpaths. The only method left for implementation is the `launch` method, which is invoked by Eclipse when a user runs UnitCheck with a certain configuration. The method creates a new runner (`UnitCheckEclipseRunner` described before) and uses `RunContext` to run it.

Figure 4.4 Launch Delegate and Launch Shortcut



The situation with `UnitCheckLaunchShortcut` is very similar, it extends the class with four abstract methods.

- `getConfigurationTypeName()` – returns the identifier of a configuration type as was defined in the `plugin.xml` file.
- `isGoodMatch(ILaunchConfiguration c)` – decides whether an existing configuration can be used by the launch shortcut (based on what a user selected to run).

- `initializeConfiguration(ILaunchConfigurationWC c)` – initializes a new configuration (used when no existing suitable configuration was found).
- `launch(ISelection s, String mode)` – the entry point to the shortcut. First, it finds an existing configuration or creates a new one using the previous methods. Afterwards, the configuration is launched which subsequently launches the corresponding delegate (`UnitCheckLaunchConfigurationDelegate`).

The `UnitCheckLaunchConfigurationTabGroup` class is an extension that creates a dialog for editing UnitCheck configurations. The dialog comprises of a set of tabs, most of them are standard tabs provided by the `org.eclipse.ui.debug` plugin (e.g., for setting up a classpath and sourcepath).

4.3.3 GUI – Views and Preference Page

Eclipse uses Standard Widget Toolkit (SWT) [18] for making up the graphic user interface. SWT is a Java toolkit that uses native objects (e.g., GTK+ objects) which is in contrast to Swing that draws its own widgets. As a result, SWT applications always keep the look of the host system.

JFace [12] is a set of helper classes that make the life with SWT easier for a programmer. It provides viewers for displaying, sorting, and filtering plain Java objects in list views and tree views. JFace also allows to implement actions and assign them to menu items and buttons. When using JFace, programmers do not have to solve low-level UI problems but they can concentrate on the actual functionality.

The UnitCheck plugin provides two views, implementations of the `org.eclipse.ui.views` extension point described at the beginning of the Eclipse section. The result tree view (`ResultTreeView`) presents the tree of test classes and test methods checked in the latest run of the plugin. The path view (`PathView`) shows detailed information about the selected test method. Both views extend the `ViewPart` abstract class with three important abstract methods:

- `createPartControl(Composite parent)` – receives the parent UI object, creates the UI of the view (buttons, labels, viewers, layout, ...), and assigns actions to the view's toolbar.
- `setFocus()` – called when the view receives the focus.
- `dispose()` – called when the view is disposed, the necessary cleanup is performed here.

The views are accessed on multiple places of the plugin's code. The `ViewRegistry` class is used to make the access easier. It provides methods for

- registering newly created instances of the view classes,
- getting existing instances of the view classes,

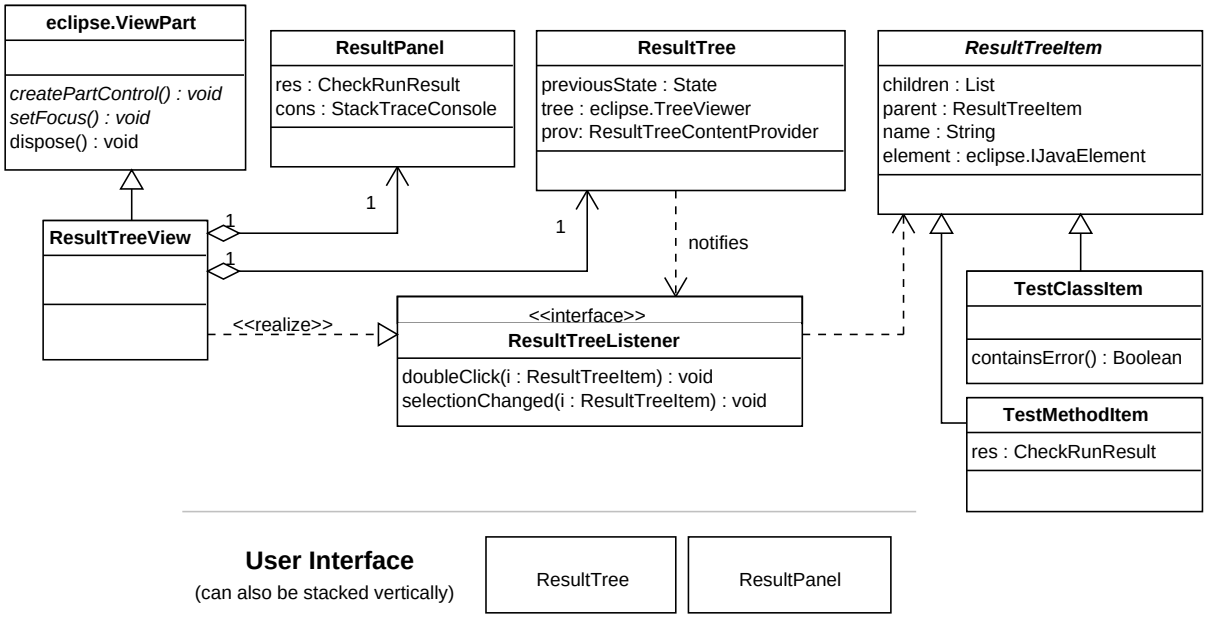
- adding and removing listeners, which are called when a new instance of a particular view is registered.

Besides the views and the launch configuration dialog, the plugin creates one more GUI item – the preference page which is a standard Eclipse solution for setting global plugin's preferences. The page is an extension of the `org.eclipse.ui.preferencePages` extension point and is implemented in the `UnitCheckPreferencePage` class.

4.3.4 Displaying the Results of Checking

The previous section described the views in general. Now the focus will be given on how the results of checking are displayed in the views. The first is `ResultTreeView` (Figure 4.5) which is split into two parts – `ResultTree` contains the tree of checked test classes and test methods, `ResultPanel` shows the summary of results for the selected test method and buttons for displaying more detailed information (e.g., an exception stack trace).

Figure 4.5 `ResultTreeView` – Summary of Checking Results



`ResultTree` uses JFace `TreeViewer` to display the tree of results. Each viewer needs a content provider which populates it with data and a label provider which returns a label and icon for each item in the viewer. In this particular case, the `ResultTreeContentProvider` class is filled with checking results by `UnitCheckEclipseListener`. The content provider transforms the data provided by the listener into its own representation suitable for the tree viewer:

- `ResultTreeItem` – abstract class representing the tree character of a node (parent node, list of child nodes). It also holds the corresponding Eclipse reference to a Java class or method.
- `TestClassItem` – represents a test class.

- `TestMethodItem` – represents a test method and holds the result of checking the method.

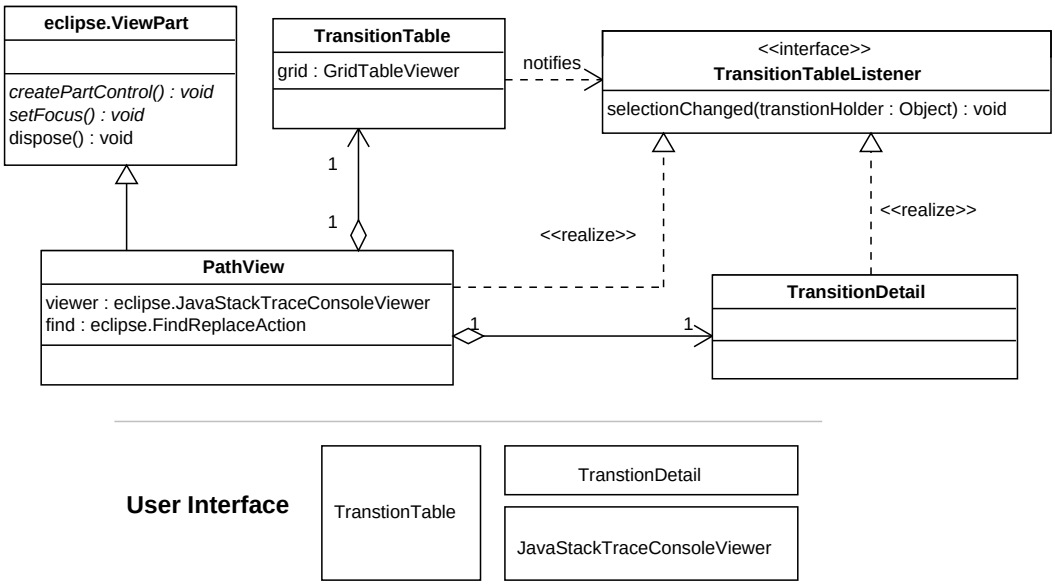
The tree also keeps its previous state of expanded and collapsed nodes, so that after rerunning the latest launch configuration, the tree looks same. Other components of the plugin may register themselves with the result tree to be notified when an item in the tree is selected and double-clicked. As an example, `ResultTreeView` implements `ResultTreeListener` and opens the Eclipse Java editor when an item in the tree is double-clicked.

`ResultPanel` is not very complicated class. A nice feature is the ability to print the detailed information about errors found during checking to the Eclipse Stack Trace Console. As a consequence, users can see highlighted stack traces and execution histories, which are clickable and open the corresponding code locations in the Java editor.

The second view (`PathView`) is useful only when an error was found while checking a test method. It shows the complete *execution path* leading to the error (the term *execution history* is often used, too). The path is split into transitions which are pieces of code executed by one thread. Figure 4.6 shows the structure of the view – it is composed of three components:

- `TransitionTable` – shows a list of transitions in a grid[3]. Same as other JFace viewers, the grid employs a label and content provider to display the transitions. The table notifies other components of the plugin when a transition is selected.
- `TransitionDetail` – shows detailed information about the selected transition.
- `JavaStrackTraceConsoleViewer` – uses `JavaStrackTraceConsole` to print the code executed in the selected transition.

Figure 4.6 `PathView` – Details of Found Errors



[3] `GridTableViewer`, which is a part of the Nebula project [14], is used. Soon, it will be merged in JFace.

To search the code listings printed in the console viewer, the `FindReplaceAction` class from the `org.eclipse.ui` plugin is used.

4.3.5 Distribution of the Plugin

Eclipse uses the concept of plugins, features, and update sites for distribution and installation of contributions.

Plugin
: This is the basic building block in the Eclipse platform. Plugin is a package of code and resources that together form a piece of functionality. As an example, everything we have seen so far forms one plugin.

Feature
: In some situations, it is useful to split the functionality into a couple of plugins that work together. Such fine-grained plugins then expose extension points that are used by other plugins. It allows to reuse the plugins and replace them with alternative implementations. More plugins can together form a *feature* which is distributed, installed, and updated as a complex unit. The `feature.xml` file lists a collection of plugins that belong to the feature and describes the feature (license, copyright).

Update site
: Update sites allow to deploy features on client machines. Each site is described in the `site.xml` file which lists all the features that can be installed using the site. The features and corresponding plugins are located in the `plugins` and `features` directories in the form of JAR files. Update sites can be remote (features are installed over HTTP) or local (features are installed from a local directory or a ZIP file).

Currently, UnitCheck consists of only one plugin. A possibility of splitting it into more plugins was also considered. The plugin could be broken into two basic parts, the UI and non-UI part, which would allow to build an alternative user interfaces for UnitCheck. Moreover, the UI part could be further split into more general parts, which would enable to use the same UI also for a simple JPF plugin (with no JUnit employed). For simplicity, a decision was made to start with only one plugin.

The single UnitCheck plugin is included in a feature, which can be installed both remotely and locally.

Chapter 5

Case Study

This chapter presents how UnitCheck can be used for checking correctness of a few features in a small application. An emphasis is given on the description of the benefits which UnitCheck brings in the verification process of the application. The test suite comprises of a set of JUnit test cases and custom JPF properties that together form the input of the UnitCheck tool. In this text, two JUnit test cases and one custom property are thoroughly described.

5.1 Daisy Filesystem

The Daisy filesystem [8] is used as an example of an application that can be successfully checked with the tools described in the previous chapter. It is important to note that Daisy is not a real world filesystem. Initially, it was used as an input program in a contest for various testing and validating tools (Java PathFinder, Bandera, etc.). This is also the reason why the filesystem intentionally contains a bunch of bugs that were to be detected using various techniques including model checking and run-time analysis. The filesystem is highly concurrent so the tools especially have to show their abilities when dealing with concurrent applications.

Daisy meets both preconditions put on the programs that can be checked with the UnitCheck tool – it is written in Java and it is not very large, just 1KLOC. A number of JUnit 4 test cases and JPF properties were written that test all important filesystem calls and try to find as many errors as possible in the implementation.

5.2 Testing Environment

Daisy has a form of a Java library that can be embedded in other Java applications. To create a testing environment, the filesystem is embedded in each JUnit test case through the common base class `AbstractDaisyTest` (Listing 5.1). The class is in charge of the filesystem initialization performed in the `init` method, which is marked with the JUnit `Before` annotation. Therefore, it is invoked by JUnit before any test method is called.

Each concrete test case contains one or more methods, marked with the JUnit `Test`

annotation, that perform a particular test of a Daisy component or feature. Such JUnit tests can be run directly using a JUnit runner or in a JUnit IDE plugin. The chance of locating bugs in a concurrent environment increases a lot when unit checking, provided by UnitCheck, is employed for the verification of Daisy. UnitCheck analyzes all thread interleavings applicable in each test.

Listing 5.1: Base Class of All Daisy Tests

```
/** Abstract class for all Daisy tests. The class provides tests
 *  with filesystem initialization and a few member fields that
 *  are commonly used in test cases. */
public abstract class AbstractDaisyTest {
  protected FileHandle root = new FileHandle();
  protected FileHandle fh = new FileHandle();
  protected Attribute a = new Attribute();

  // Run automatically by JUnit before any test method is called.
  @Before
  public void init() {
    root.inodenum = 0;
    Petal.init(false);
    System.out.println("Disk contents:");
    Daisy.dumpDisk();
  }
}
```

An example of a simple JUnit test case follows. A more complicated test case is described in the next section. Listing 5.2 shows a piece of the test that is part of the original Daisy package. The goal of the test is to call basic filesystem methods for creating, deleting, and writing files. After each operation is finished, the filesystem contents is printed to check whether the operation was successful or not. For simplicity, deleting and writing was omitted from the example.

Listing 5.2: Simple Daisy Test

```
/** Original test shipped with Daisy distribution. */
public class OriginalTest extends AbstractDaisyTest {
  @Test
  public void test() {
    System.out.println("Creating file named cow in root:");
    DaisyDir.creat(root, DaisyTest.stringToBytes("cow"), root);
    System.out.println("Return code: "
      + DaisyDir.creat(root, stringToBytes("cow"), fh));
    System.out.println("Returned file handle: " + fh);
    System.out.println("Disk contents:");
    Daisy.dumpDisk();
  }
}
```

Java Pathfinder, which provides UnitCheck with the model checking capability, allows to implement custom properties that must hold in an errorless program. In other words, a violation of a property indicates a bug in the program. When it happens, UnitCheck provides a complete execution history leading to the violation. Some features of the filesystem implementation are too complicated to be tested by traditional JUnit tests.

In such case, JPF properties, which allow to track every detail of the test execution, are implemented and JUnit tests provide an environment for checking validity of these properties in Daisy.

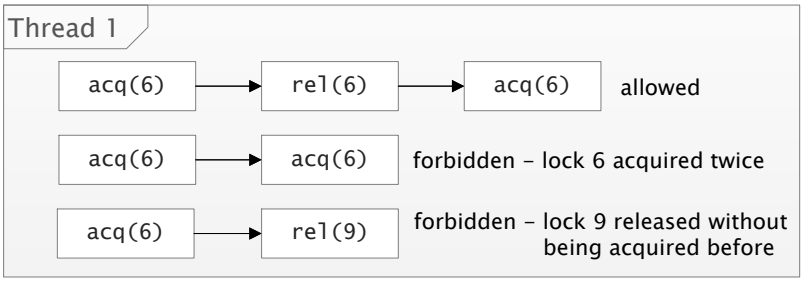
5.3 Complex Test Case

A couple of Daisy properties, which should be checked by the tools used for the verification of Daisy, were proposed in [8]:

- Correct usage of locks. Each thread should acquire and release each lock in a strict alternation.
- Deadlock freedom. Prove that locks are acquired by threads according to a global partial order.
- Invariants on data structures of the file system. If an allocated inode points to a block, then that block must be allocated as well, etc.

The first property was selected for demonstrating the power of the UnitCheck tool. First, it is important to make clear what the property allows and what not. Figure 5.1 shows sequences of lock operations[1] that are allowed and forbidden assuming that in each row, the first operation from left is also the first `acq`/`rel` operation called within an instance of the Daisy filesystem.

Figure 5.1 Daisy Lock Sequences – Allowed and Forbidden



To check the property with UnitCheck, two steps have to be done.

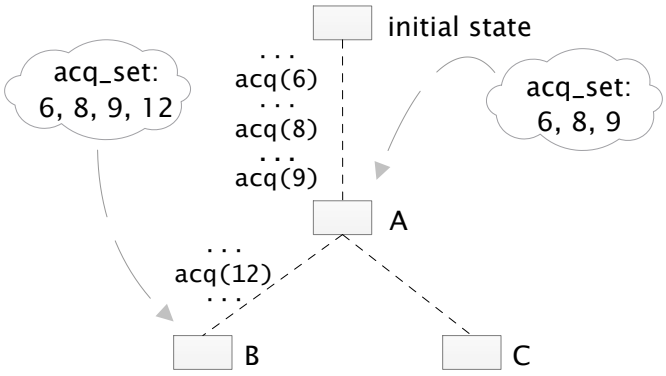
1. A JPF property is implemented. It monitors every invocation of the `LockManager` `acq`/`rel` methods and reports a failure when a forbidden sequence appears in some thread.
2. A series of JUnit tests is written to simulate situations where `LockManager` is extensively called.

[1] `acq(long n)` acquires the lock number `n`; `rel(long n)` releases the lock number `n`.

In order to check the alternation of lock operations, the property stores the state of all locks for each thread – whether they have been acquired or released in the thread. What causes a complication is the JPF backtracking mechanism illustrated in Figure 5.2.

Let us suppose that UnitCheck, while analyzing the Daisy filesystem, reaches the state A in which a transition point is created with two branches. The thread number 1 has the locks 6, 8, 9 acquired in the state A. Before the state B is reached by the transition from A, one more lock number 12 is acquired by the same thread and this information is stored in the property. After backtracking into the state A, we are no more interested in what happened with the locks on the transition to the state B. The state of all locks has to be restored so that the transition to the state C starts with the same information as the transition to the state B started. Therefore, the property manages a log about acquired and released locks. The information is kept in a stack and when JPF backtracks, the stack is used for rollbacking the states of the locks that are no longer acquired or released.

Figure 5.2 Restoring Lock Information When Backtracking



* all operations and acq_sets apply for one thread (e.g., thread number 1)

Listing 5.3 shows the implementation of the property described in previous paragraphs (technical details were omitted). The property uses a convenient JPF mechanism for creating properties called `PropertyListenerAdapter` which combines JPF listeners with a property interface. The most important are three overridden methods – `stateAdvanced` and `stateBacktracked` are in charge of keeping the log of lock operations as it was described above. The `instructionExecuted` method monitors the invocations of the lock methods (`acq`, `rel`) and checks whether each lock is acquired and released in a strict alternation.

Listing 5.3: `LockOrderProperty` – Monitors Lock Operations

```
/** Property monitoring acq/rel operations. */
public class LockOrderProperty extends PropertyListenerAdapter {
  private static final String LOCK_ACQ_METHOD = "LockManager.acq";
  private static final String LOCK_REL_METHOD = "LockManager.rel";
  private static final String LOCK_NO_ARGUMENT = "lockno";

  private LockData locks = new LockData();

  @Override
  public void instructionExecuted(JVM vm) {
```

```
  Instruction instr = vm.getLastInstruction();
  ThreadInfo ti = vm.getLastThreadInfo();
  if (instr instanceof InvokeInstruction) {
    if (LOCK_ACQ_METHOD.equals(getMethodName(instr))) {
      Long lock = instr.getArgumentValue(LOCK_NO_ARGUMENT, ti);
      checkLockAcq(ti.getIndex(), lock);
    } else if (LOCK_REL_METHOD.equals(getMethodName(instr))) {
      Long lock = instr.getArgumentValue(LOCK_NO_ARGUMENT, ti);
      checkLockRel(ti.getIndex(), lock);
    }
  }
}

@Override
public void stateAdvanced(Search search) {
  locks.stateAdvanced(search.getStateNumber());
}

@Override
public void stateBacktracked(Search search) {
  locks.stateRestored(search.getStateNumber());
}

private void checkLockAcq(int threadIndex, long lock) {
  if (locks.isLocked(threadIndex, lock)) {
    // report violation!
  }
  locks.lockAcq(threadIndex, lock);
}

private void checkLockRel(int threadIndex, long lock) { ... }

/** Represents a single invoked lock operation (acq, rel). */
private static class LockOperation {
  protected enum OP { ACQ, REL };
  protected final OP op;
  protected final int thread;
  protected final long lock;
}

/** Stores the current state of locks. */
private static class LockData {
  /** State of locks for each thread (key=threadIndex). */
  private Map<Integer, Set<Long>> lockStates;
  /** History of acq/rel/stateAdvanced events. */
  private Stack<Object> history;

  public boolean isLocked(int thread, long lock) { ... }

  public void lockAcq(int thread, long lock) {
    getLockStates(thread).add(lock);
    history.push(new LockOperation(LockOperation.OP.ACQ,
                                   thread, lock));
  }

  public void lockRel(int thread, long lock) { ... }
```

```
    public void stateAdvanced(Integer state) {
      history.push(state);
    }

    public void stateRestored(Integer state) {
      while (!history.isEmpty()) {
        Object last = history.peek();
        if (last instanceof Integer) {
          if (last.equals(state)) {
            break; // leave while
          }
        } else if (last instanceof LockOperation) {
          LockOperation op = (LockOperation) last;
          switch (op.op) {
          case ACQ:
            getLockStates(op.thread).remove(op.lock);
            break;
          case REL:
            getLockStates(op.thread).add(op.lock);
            break;
          }
        }
        history.pop();
      }
    }
  }
}
```

Listing 5.4 contains one of test cases that may serve as a testing environment for checking the validity of `LockOrderProperty`. This test case focuses on testing the `DaisyDir.creat` method in different situations.

1. `testCreatFile` creates a single file.
2. `testCreatFileThreads` creates more files in a couple of threads. All thread interleavings are analyzed by UnitCheck.
3. `testCreatLongFilename` creates a long filename.

   It is the only test within the test case that violates `LockOrderProperty`. Daisy allows to create filenames up to 256 characters, otherwise it throws an exception. The lock acquired in the first `creat` call is not correctly released on error. Therefore, UnitCheck reports a violation of the property when Daisy tries to acquire the same lock again in the second `creat` call.

Listing 5.4: Test of the Daisy `creat` Method

```
/** Tests the creat method. */
public class CreatFileTest extends AbstractDaisyTest {

  private static final int LONG_NAME = 257;
  private static final int THREADS = 2;
```

```
  @Test
  public void testCreatFile() {
    creatFile("xyz");
  }

  @Test
  public void testCreatFileThreads() {
    for (int i = 0; i < THREADS; ++i) {
      Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
          creatFile("t_" + j);
        }
      });
      t.start();
    }
    // ... join all threads ...
  }

  @Test
  public void testCreatLongFilename() {
    String filename = TestUtil.createFilename(LONG_NAME);
    try {
      creatFile(filename);
    } catch (Exception e) {
      e.printStackTrace();
    }
    creatFile("abc");
  }

  private void creatFile(String name) {
    System.out.println("Creating file named " + name
      + " in root:");
    System.out.println("Return code: "
      + DaisyDir.creat(root, stringToBytes(name), fh));
    System.out.println("Returned file handle: " + fh);
  }

}
```

Chapter 6

Related Work

A significant amount of related work was listed in the introduction to unit checking in Section 2.3. A focus was given on the theoretical background of unit checking and various ideas that integrate unit testing and model checking. This section focuses more on existing tools that are somehow related to unit checking, novel testing techniques, or to the UnitCheck implementation. A special focus is given on tools employing JUnit and JPF, and tools targeting the Java programming language.

6.1 Testing Frameworks

JUnit, which is a part of UnitCheck, comes from the family of xUnit testing frameworks. They are all very similar – the main difference is mostly in the target programming language. Therefore, these frameworks are not listed here. Instead, the testing frameworks which employ a model checker (Section 6.1.1) or automatically generate a suite of regression (JUnit) tests (Section 6.1.2, Section 6.1.3) are described here.

The papers listed in Section 2.3 often propose to use a model checker for generating test cases. Some of the test-generating tools mentioned here use ideas related to model checking (e.g, Pex), some not (e.g., JCrasher) and some do not provide their implementation details (e.g., Jtest). Agitar and Pex are described later in more detail because they are used in the industry and enough information about them is available.

The jCUTE tool [61] uses concolic execution[1] to handle concurrency and generate JUnit tests. The jFuzz tool [34] employs JPF to perform concolic execution and generate inputs that exercise new program paths. JCrasher [31] uses random input data to exercise the Java code and create tests. Jtest [16] is a tool for generating a suite of regression JUnit tests. Additionally, it checks a huge set of coding rules to prevent frequent errors. CoView [7] generates only the stubs of JUnit tests which have to be completed by developers. Besides generating regression JUnit tests, CodePro AnalytiX [5] has a support for design by contract.

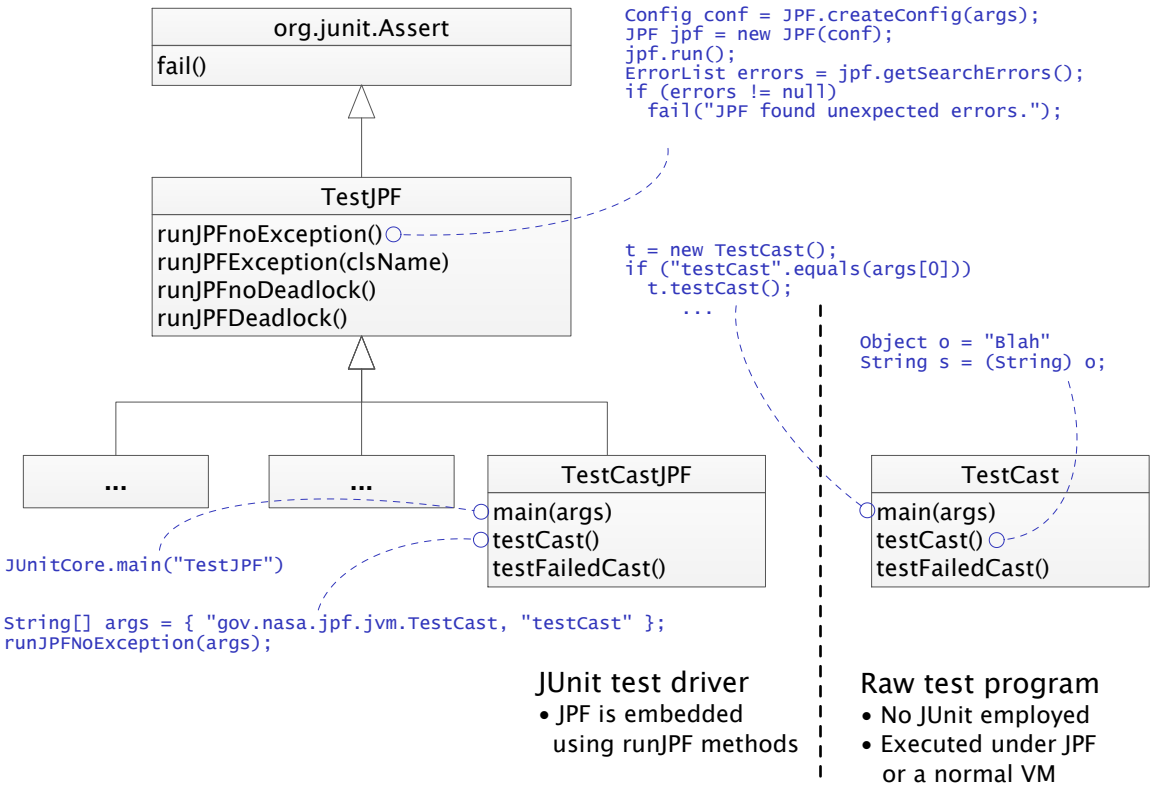
[1] Concolic execution simultaneously combines concrete and symbolic execution of the code under test.

6.1.1 Java PathFinder Test System

JPF itself comes with its own test system used for regression testing of two major JPF components – a virtual machine (class loading, field access, etc.) and a model checker (state management, search strategies, etc.). The JPF test system is not an example of a generic testing framework. It is mentioned here because both tools (JPF and JUnit) integrated in UnitCheck are used in this system, too. Differences between UnitCheck and the JPF test system, their capabilities, and typical use cases are covered in this section.

The design of the JPF test system results from the need for running tests in a number of ways – under a normal VM, under JPF itself, and using JUnit. Figure 6.1[2] shows the structure of tests and one example of a test. Each test comprises of two classes.

Figure 6.1 Java PathFinder Test System – Structure of Tests



Raw test program (`TestCast`)
: This is a plain Java executable program containing the test itself. It comprises of a number of test methods performing the actual test work. The `main` method always follows the same scheme – it only calls a test method whose name was passed as an argument. These test programs can be executed under JPF or a normal virtual machine.

JUnit test driver (`TestCastJPF`)
: The driver is a simple wrapper above the corresponding raw test class. It adds a

[2] Taken from `http://javapathfinder.sourceforge.net`.

support for running the raw tests under JUnit. For each test method of the raw test class, there is a method marked with the JUnit `@Test` annotation in the driver. This JUnit test just runs the corresponding raw test under JPF using the `JPF` class (see 3.2 for details about embedding JPF) and one of the `runJPF...` methods which checks whether the raw test finished as expected or not. In addition, the test driver contains the `main` method which explicitly calls the JUnit console runner.

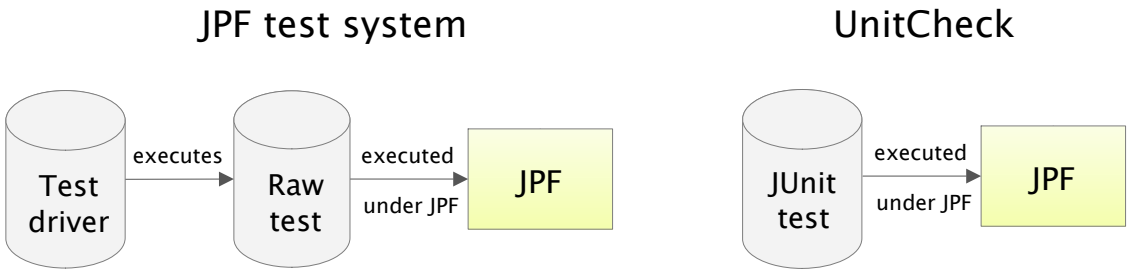
The JPF test system is used for testing JPF itself. JUnit facilitates running of a number of raw tests in a batch and provides reports about the count of successful and failed tests. There is no support for running our own JUnit tests under JPF using this system. It requires a bit of extra non-testing code (test drivers and `main` methods of raw tests) to be written which is in contrast to ordinary JUnit tests. On the other hand, the test system allows to execute tests without JUnit.

On the contrary, UnitCheck is a result of an effort to run ordinary JUnit tests under JPF without losing a possibility to run them under a normal virtual machine. Test writers do not have to learn anything new – tests executed under UnitCheck are pure JUnit tests. Even the way of running tests is very similar to JUnit runners and plugins. The users of UnitCheck may benefit from all advantages of the JPF model checker described in the previous chapters.

It cannot be said which system is better. Both of them have different objectives, especially in the way of employing JUnit which is once more illustrated in Figure 6.2. The JPF test system cannot be completely replaced with pure JUnit tests executed under UnitCheck. There are two reasons why.

- In UnitCheck, property violations are always treated as errors. The JPF test system allows to create tests that expect a violation of a property (e.g., `runJPFDeadlock`).
- Besides exceptions raised by raw tests, the JPF test system handles also exceptions thrown by JPF itself.

Figure 6.2 Java PathFinder Test System vs. UnitCheck



6.1.2 Agitar

Agitar [1] comes with two techniques for improving the process of testing Java applications. First, it automatically generates a suite of JUnit tests. For each class in the application

being tested, a JUnit test case is generated with a lot of test methods covering as much code of the class as possible. These regression tests are not intended to be read and edited by developers. The tests can be run later to assure that changes in the implementation do not change the behavior of the application.

To resolve dependencies in tests, mocking is used by Agitar. The generating of tests can be directed by developers. *Test helpers* are classes that help Agitar to achieve higher code coverage by enumerating arguments used for invoking tested methods. *Class invariants* specify contracts that are checked in every generated test.

The second technique is called *agitation*, a kind of *exploratory testing* [29]. Agitar exercises the code with various input data and presents a list of observations. If an observation describes an intended behavior of the code, the developer may direct Agitar to create an assertion (JUnit test) from the observation. Otherwise, the developer may immediately fix the bug.

Agitar is a client-server system. The client part is integrated into Eclipse with plugins for generating tests, running agitation, etc. As the generating can be very time-consuming for bigger projects, it is performed by the server which can be spread on multiple machines. The server also provides reports about code coverage and complexity, and has a support for continuous integration and testing.

6.1.3 Pex

Before digging deeper in Pex, let us describe the concept of *parametrized unit tests* (PUTs) [62] which are extensively used by Pex. A traditional object-oriented unit test is parameterless, it executes the code under test with fixed arguments hard-coded in the test method. On the other hand, a parametrized test is generic, it uses varying inputs (parameters) for executing the code under test. A traditional unit test can be viewed as an instance of a parametrized test.

Testing frameworks use different approaches to inject the actual values of parameters in tests. In JUnit, parameters are represented as member fields in test classes, filled by so called parameter feeders, and a special runner (`Parametrized`) has to be used for running them. In Pex, it is more straightforward – test methods have arguments that represent the test parameters.

As a result, these generic parametrized tests can be run multiple times with various parameters. In a traditional unit test, it can be achieved only by looping over all values of the parameters (either by an explicit loop in a test, or by creating a new test for each value). Listing 6.1 illustrates differences between both types of tests that are testing the `Trim` method. The first test uses only one fixed string as an input, while the second takes the input as a parameter.

Listing 6.1: Traditional Unit Test vs. Parametrized Test

```
[TestMethod]
void TestTrim() {
  String s = "Hello   ";
  String res = s.Trim();
  Assert.isTrue("Hello" == res);
}
```

```
[PexMethod]
void TestTrim([PexAssumeNotNull] String s) {
  String res = s.Trim();
  // assert that all leading and trailing characters are removed
  Assert.isTrue(...);
}
```

The question is what values should the testing framework use for the parameters of tests. In JUnit, the values are explicitly specified in a test by a programmer. Therefore, the JUnit parametrized tests (or data-driven tests) are only a syntactic sugar for testing code with more input, still fixed, data sets. Pex comes with a different approach, it automatically generates the (minimal) set of inputs that (in ideal case) fully cover the code reachable from the test.

Pex [47, 53, 55, 48] is a Microsoft tool integrated into MS Visual Studio[3]. The tool is not restricted to the usage of parametrized unit testing. It can analyze any .NET method, generate its interesting inputs, and display the output for each generated input (if an exception is raised for a certain input, its stack trace is provided). The generated inputs can be saved as a test suite in the form of traditional unit tests[4] and used for debugging the failed inputs and for regression testing (both without Pex being involved).

The generated test cases provide high code coverage [69] (sometimes even 100%). Therefore, they can reveal null pointer dereferences and violations of contracts and assertions used within the actual implementation. But the generated tests do not check that the method being tested behaves as expected. To check the behavior of methods, Pex runs parametrized unit tests written by programmers. PUTs specify assumptions to filter out some inputs and assert the desired behavior of methods under test (see Listing 6.1).

To achieve high code coverage, Pex does not select input values randomly. Instead, it employs dynamic symbolic execution. Pex executes the code multiple times and searches for branches that were not covered previously. If such a branch is found, a constraint system (i.e., a predicate over the test inputs) is constructed and the Z3 constraint solver [21] tries to find inputs that lead to reaching that branch. If it succeeds, the code is executed with the new input and the whole process is repeated.

We believe that features provided by all the test-generating tools mentioned above (including Pex) can be used in synergy with unit checking. Automatically generated tests can be run under UnitCheck and users will benefit from exploring all admissible thread interleavings[5].

6.2 IDE Integrations

This section presents model checking tools integrated in various IDEs with a focus on Eclipse which is used for the UnitCheck plugin. Two plugins that bring support for run-

[3] Command-line interface is also provided to allow for automation.

[4] Pex integrates with MSTest, xUnit, NUnit, MbUnit, and other testing frameworks.

[5] Besides Pex, Microsoft offers CHESS [57] – a tool for finding and reproducing bugs in concurrent programs.

ning Java PathFinder within the Eclipse IDE[6] are covered in more detail. Although a direct comparison with the UnitCheck tool cannot be made, it is important to mention these plugins. Same as UnitCheck, they integrate Java PathFinder in Eclipse, the difference is that UnitCheck adds support for running JUnit tests under JPF. But from the user point of view, the UnitCheck plugin and other plugins offer similar features and UI which can be compared. Moreover, the evaluation of other JPF plugins is important for future considerations of extending the UnitCheck plugin with a feature to run pure Java applications under JPF (as other plugins do).

There are Eclipse plugins for other model checkers – BLAST [23] and CMBC [4] targeting C programs, Spin [63, 59] for models written in Promela, Bogor [3] using its own modeling language, TLC model checker [19] which uses TLA+ specifications.

6.2.1 JPFep[7]

JPFep [10] is a short for Java PathFinder Eclipse Plugin. The project is developed at the Technion and describes itself as a *plug-in that integrates JPF into Eclipse as a special kind of debugger and provides a user friendly interface for configuring JPF's large number of parameters.*

The plugin has not been updated since 2008 which is probably the reason why it does not work with the latest version of Java PathFinder[8] and also one of the reasons why it was impossible to successfully run the whole plugin. Nevertheless, some parts of the plugin are working or at least partially working which allowed to summarize a couple of user comments.

+	An internet update site allows to install the plugin in an easy and convenient way.
+	Debug configurations and launch shortcuts are standard Eclipse solutions that plugins running various resources should have.
+ –	A lot of JPF parameters can be specified through the GUI. On the other hand, no default values are provided for such parameters and it takes a lot of time to fill them correctly.
–	JPF is not bundled as a part of the plugin. Users have to bother with installing and configuring JPF. Changes in JPF may even break the plugin.
–	Debug perspective is used by the plugin which is a little bit confusing when there is no chance to use breakpoints or track variables in a program checked by JPF. Java perspective and Run configurations would be better for the purpose of this plugin.

JPFep was originally a part of the CAPE tool suite [6] which is an environment for various verification and analysis tools. A plugin that integrates JPF into Eclipse is one

[6] In [40], authors describe the implementation of another Eclipse plugin. A plugin for the NetBeans IDE is developed in the main trunk of Java PathFinder. As this plugin has not matured yet, it is not thoroughly evaluated here.

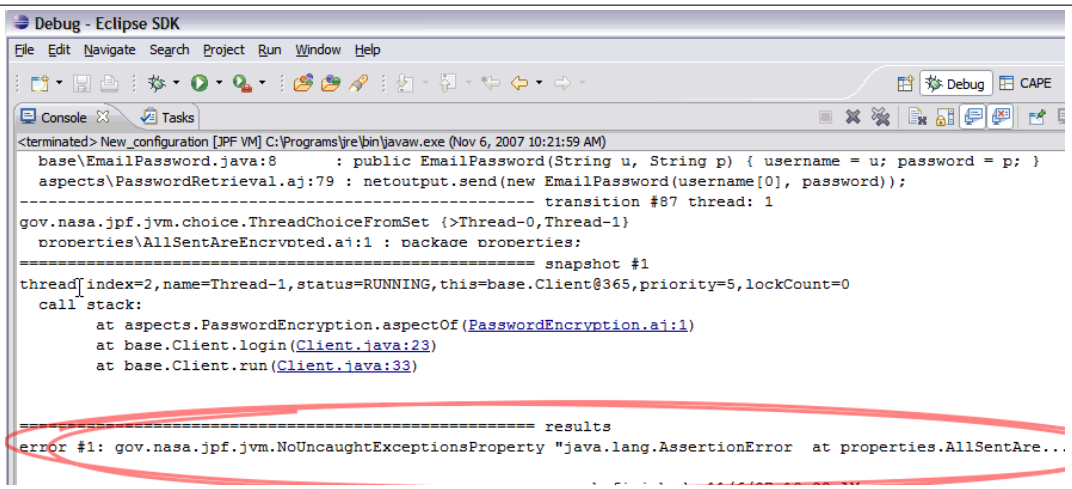
[7] The following text applies to the version 1.1.0.

[8] There have been a number of changes that made older JPF extensions incompatible with the latest JPF (e.g., `vm.peer_packages` has been renamed to `vm.peer.packages`).

of subprojects in this suite. Unfortunately, everything that has been said so far about JPFep applies also for this older plugin which is not under active development. A couple of comments can be added based on a video tutorial that can be found on the CAPE's project page.

–	No way how to configure JPF parameters in Eclipse.
–	JPF output is displayed only in the Console view (see Figure 6.3). Therefore, the plugin does not come with much added value in comparison with the command-line JPF tool.

Figure 6.3 JPFep – Displaying Results of Checking



6.2.2 Visual Java PathFinder[9]

Visual Java PathFinder (VJP) [20] is another plugin that integrates JPF in Eclipse. The project was started in 2008 during Google Summer of Code by Sandro Badame and is still actively developed. A quotation from the project's page follows.

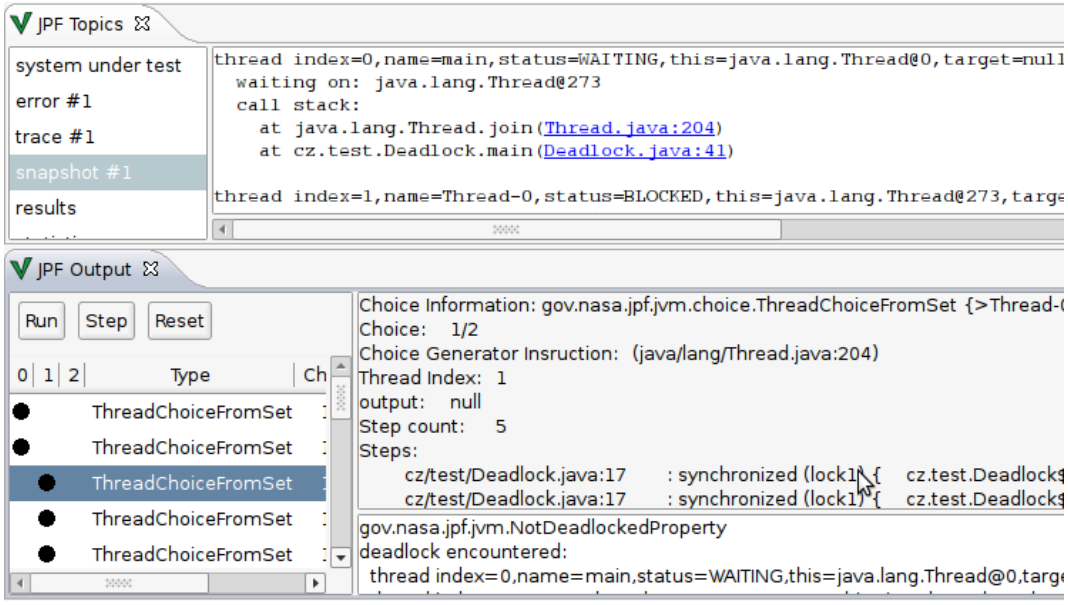
> *VJP serves to simplify the process of verifying your program. Verifying Java programs with JPF requires downloading, updating, building, configuring and verifying, all manually. On the other hand with VJP handles all of this and allows configuring and verifying to be done through a GUI with the push of a button.*
>
> *On top of all of this, one of VJP's main advantages is that it also displays JPF's results graphically. Execution paths are graphically displayed giving the details of every step taken.*

Same as with the previous plugin, the following table contains some of the plugin's pros and cons.

[9] The following text applies to the version 1.0.19.

+	An internet update site allows to install the plugin in an easy and convenient way.
+	JPF is packaged with the plugin and therefore, users do not have to install and configure it.
+	JPF configuration parameters can be set in the GUI with a possibility to use the default values. Configurations can be saved in files and used with standalone JPF.
+ −	Topics view (see Figure 6.4) uses the JPF `Publisher` interface and highlights links to source codes. On the other hand, it only displays what the JPF command line tool prints, without any graphical representation.
+ -	Output view (see Figure 6.4), which is no longer the default one and has to be explicitly configured, splits the output information into a set of transitions and displays them separately. The execution trace of each transition is not clickable and the view as a whole does not fit Eclipse very well.
−	Verify view completely duplicates Eclipse Run configurations view which should have been used instead.
−	Eclipse launch shortcuts are not implemented.

Figure 6.4 VJP Views – Topics, Output



Chapter 7

Summary and Conclusion

This work presented the idea of unit checking which brings the benefits of code model checking to the unit testing area the developers are familiar with. The prototype implementation called UnitCheck integrates JUnit testing framework and Java PathFinder model checker. The UnitCheck tool allows to run standard JUnit tests under JPF and summarizes reports about test execution provided by both JUnit and JPF. For each test failure, a complete execution trace allows developers to quickly find reasons of the failure.

Of course, not all tests are amenable for unit checking. Only tests for which the standard testing is not complete (i.e., tests that feature random values or concurrency) would benefit from exhaustive traversal using UnitCheck[1]. As UnitCheck accepts standard JUnit tests, developers can seamlessly switch among the testing engines as necessary.

The work has reached all the main requirements listed in Section 1.2. Both JUnit and JPF are integrated in UnitCheck without changing their implementation. Only the public documented interfaces provided for extending JUnit and JPF are used by UnitCheck. The core of UnitCheck has a form of a library that can embedded in various development supporting tools. As a result, not only the Eclipse plugin allows for unit checking. The Ant task and a simple command-line application are two other user interfaces for UnitCheck.

Currently, UnitCheck has a couple of known limitations. All of them are related to Java annotations introduced JUnit 4.

- The `@Time` annotation is used in JUnit tests to assert that a test finishes within a specified time frame. JPF does not model Java time functions (e.g., `System.currentTimeMillis()`) used by JUnit. Therefore, UnitCheck does not report violations of time restrictions put on tests by `@Test` annotations.
- Tests annotated with the `@Ignore` annotation are not executed by JUnit. UnitCheck does not recognize this annotation and executes all input tests. In future, the support for the `@Ignore` annotation can be added to UnitCheck (`UnitCheckListener` will have another method called `methodIgnored`).

Although the UnitCheck tool can be used as is which is presented on the example in Chapter 5, there are still features that can be changed or added to UnitCheck in future.

[1] Only the (increasing) portion of the standard Java libraries supported by JPF can be used in the tests.

- A choice generator proposed in Section 3.3 is an elegant way of running multiple JUnit tests in one JPF run. When such a choice generator is implemented, `@BeforeClass` and `@AfterClass` JUnit annotations will behave exactly the same as in the default JUnit runners.
- The UnitCheck Eclipse plugin provides an intuitive way of running JUnit tests. It offers a user interface similar to the JUnit Eclipse plugin which is not the case of existing plugins integrating Java PathFinder in Eclipse. The UnitCheck plugin can be extended to support running plain Java applications under JPF.
- JPF comes with an extension for symbolic execution (JPF-SE) [26] of Java programs. The possibilities of employing symbolic execution to generate JUnit tests or run existing JUnit tests should be studied in future.

Besides the UnitCheck tool and this text, another outcome of the work is the paper [54] accepted for publication in LNCS.

Appendix A

User Manual

A.1 Command-line Tool

The UnitCheck command-line tool is used mainly during the development of UnitCheck. The Eclipse plugin and Ant task are a more convenient way of using UnitCheck and should be preferred.

After extracting `unitcheck-r*.zip` (or `tgz`), the UnitCheck executable programs can be found in the `unitcheck/bin` directory – `unitcheck` for Unix environments and `unitcheck.bat` for Windows. The following options are recognized, the `-t` option is required.

`-t <classname>`	JUnit test class to be checked, fully-qualified class name must be used.
`-p <path1>:<path2>:...`	Colon-separated list of classpath elements that are used for checking the test class.
`-s <path1>:<path2>:...`	Colon-separated list of sourcepath elements that are used when reporting execution history. If not specified, the history might be empty.
`-m <methodname>`	Name of a single test method to be checked. By default, all test methods of the class are checked.
`-c <filename>`	Path to a JPF configuration file. All default JPF configuration parameters are overriden with the values from this file.
`-e`	Do not print the execution history when an error occurs (by default, the history is printed).
`-a`	Trace also the standard Java libraries when an error occurs (by default, the standard Java libraries are not traced).
`-d`	Print the bytecode intructions in the execution history (by default, the bytecode is not printed).
`-j`	Do not filter the JUnit and UnitCheck code out of the execution history (by default, the filtering is on).

`-b <dirname>`	The base directory of UnitCheck. You will hardly ever need this option, it is set by wrapper scripts.
`-w`	Do not stop checking when an error occurs. It is not recommended to use this option.

The typical command for running the UnitCheck command-line program follows. A test class, its classpath, and sourcepath are set.

```
$ ./unitcheck -p /mydir/myproject/classes -s /mydir/myproject/src
    -t cz.foo.Test1
```

A.2 Eclipse Plugin

A.2.1 Installation

The plugin requires Eclipse IDE 3.4 or higher, it was successfully tested with versions 3.4.1 and 3.5M6. The plugin cannot be used with the version 3.3 or lower because it requires API that is not available in these old versions.

Before you can start using the plugin, it is necessary to properly install it to your Eclipse IDE. In order to do so, you have to access so called *update site*. This can be done in two ways:

- either you use the UnitCheck plugin internet update site[1],
- or you use the local update site archive[2].

In either case, follow these steps[3]:

1. Open the **Software Updates and Add-ons** dialog by selecting **Help**|**Software Updates** from the main menu.
2. Choose the **Available Software** tab.
3. Click **Add Site...** to define the update site.

   If you want to use the UnitCheck plugin internet update site, enter the URL of the site in the **Location** box. Click **OK** to confirm.

   In case you have the local update site archive, choose **Archive...** and in the displayed dialog select the archive file. Click **OK** to confirm.
4. Eclipse connects to the update site and offers you the features provided by the site. Check the **UnitCheck Eclipse Plugin** feature and click **Install...**.
5. In order to use the plugin, you have to agree with its license and continue by selecting **Finish**.

[1] `http://aiya.ms.mff.cuni.cz/unitchecking/plugin`
[2] `unitcheck-eclipse-plugin-r*.zip`, where `*` stands for the revision number
[3] Applies only to Eclipse 3.4, the plugin management is a little bit different in Eclipse 3.5.

6. After the plugin is installed, acknowledge the Eclipse restart.

The plugin can be uninstalled using the same **Software Updates and Add-ons** dialog (select **Help**|**Software Updates** from the main menu). In the first **Installed Software** tab, select the **UnitCheck Eclipse Plugin** item and click the **Uninstall...** button. After you confirm that you agree with the uninstallation, the plugin will be removed from your Eclipse IDE.

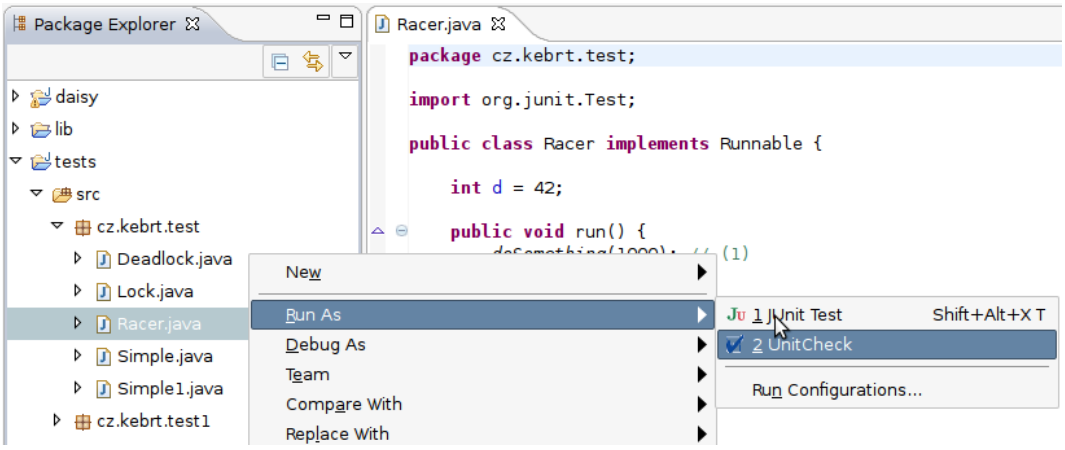
The plugin consists of two views that can be displayed by opening the **Window**|**Show View**|**Other...** dialog and selecting the views in the **UnitCheck** category. You can place the views on your favourite locations in the Eclipse IDE. Both views will be described in the next sections.

A.2.2 Running

There are a couple of ways how to run a selected JUnit test or a set of tests under UnitCheck.

- One can right click a test case, package, or a whole project in the **Package Explorer** and select **Run As**|**UnitCheck** as Figure A.1 shows. The **UnitCheck** run option is displayed only for items that contain at least one JUnit test. All test classes within the selected element are checked.

Figure A.1 Running UnitCheck from Package Explorer



- If a test class is opened in the Java editor, one may right click the source code and select **Run As**|**UnitCheck**. Only this single test class will be checked.

- The last way of running uses configurations that can be reached by selecting **Run**|**Run Configurations...** from the main menu. In the dialog, right click the **UnitCheck** category and select **New** which creates a new run configuration. The **Main** tab defines tests to be checked as Figure A.2 illustrates.

  – Either a single test class can be checked. First, a Java project must be selected using the **Browse...** button. After clicking the **Search** button, a list of all

classes from the selected project containing at least one JUnit test is displayed. Only the selected class is checked after clicking the **Run** button.

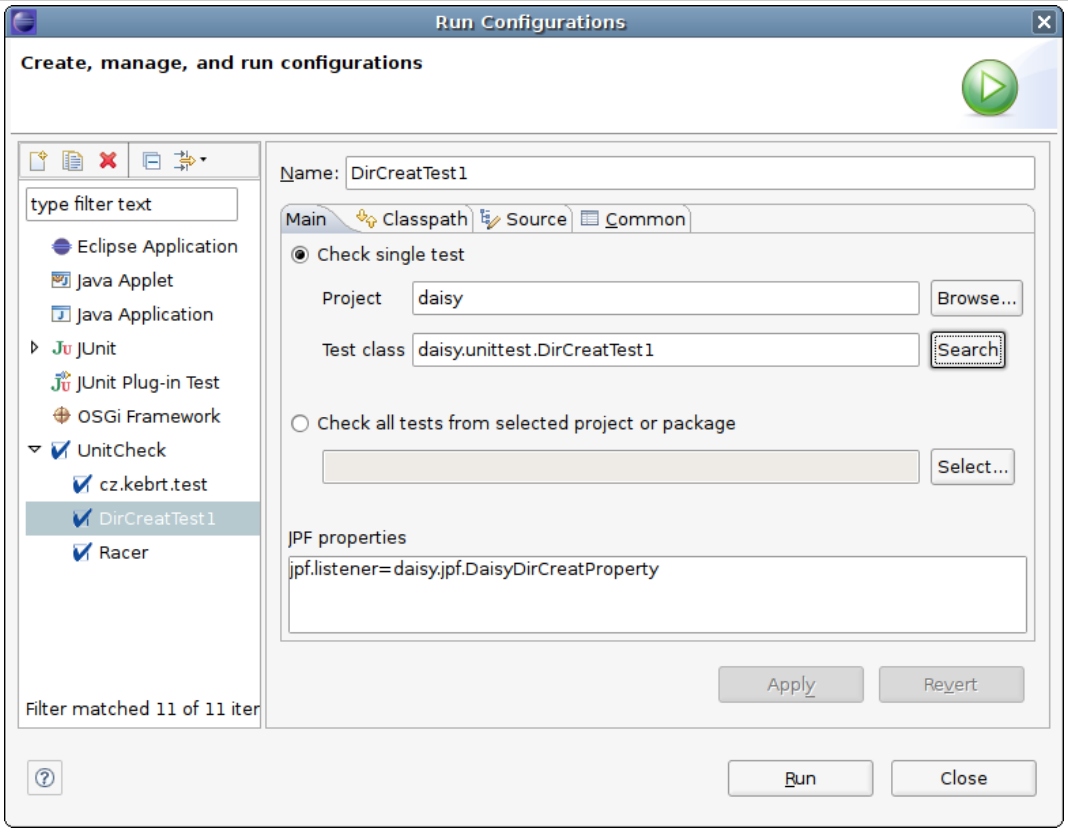
– Or all tests from the selected project or package can be checked. After clicking the **Select...** button, the tree of all projects and packages is displayed.

JPF properties overriding the default JPF properties can be specified in the textbox. Each one is written in a single row in the form of *<prop_name>=<prop_value>*. An example of a propery is `jpf.listener` for defining custom JPF listeners used when checking.

You will hardly ever need to change anything in the standard Eclipse run tabs – **Classpath**, **Source**, and **Common**. The classpath and sourcepath are based on the particular Java project's settings.

If the configuration is ok, the **Run** button starts checking of selected test classes.

Figure A.2 Running UnitCheck from the Run Configurations Dialog



A.2.3 Results of Checking

After the checking is started, the **Console** view displays the same output as the equivalent launch of the command-line UnitCheck program. The **Progress** view shows the progress of execution and allows to terminate it.

The **Check Summary** view incrementally displays the classes that were checked in the current run. The view is split into two parts as Figure A.3 shows. The first one contains

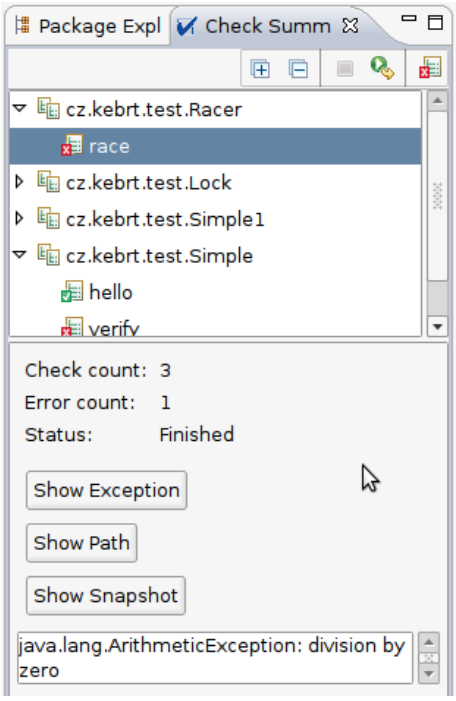
the tree of all classes that were checked. Each class comprises of a list of test methods with icons indicating whether the checking of the particular method finished successfully or not. Double clicking any item in the tree opens its source code in the Java editor.

The second part of the view contains more information about the results of checking the selected test method. In case an error was found by UnitCheck, three buttons for showing the detailed information are displayed. The first button called **Show Exception** is enabled only for errors that were caused by an uncaught exception (it is disabled for the violations of JPF properties). After clicking the button, the exception stracktrace is printed in the **Path** view which is described in the next section in more detail. The **Show Path** and **Show Snapshot** buttons allow to read the complete execution history leading to the error and the state of all threads at the moment the error occured. A short of summary of the error can be found in the textbox below the buttons.

The right upper corner of the view contains a couple of buttons for manipulating with the tree of test classes. The buttons allow to

- expand and collapse all tree items,
- stop checking,
- rerun the previous checking,
- show only the test methods that finished with an error.

Figure A.3 Check Summary View

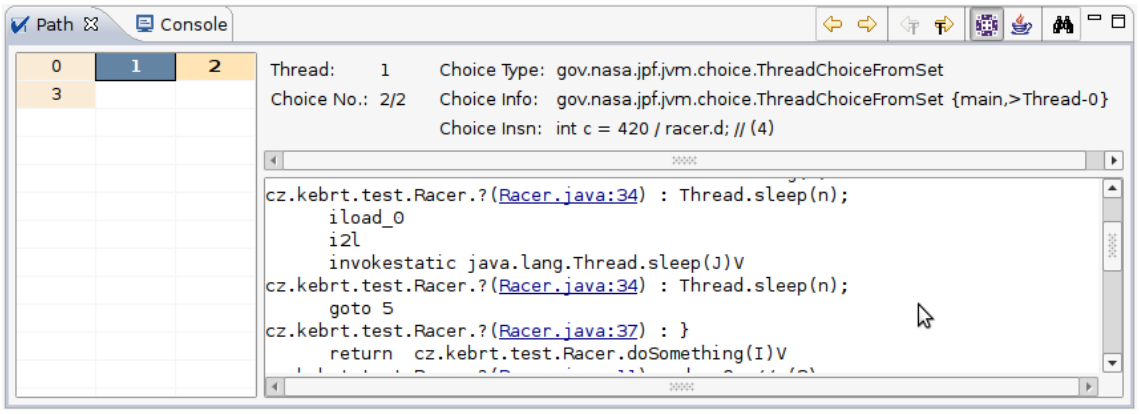


A.2.4 Inspecting Error Traces

This section applies only for situations when a test finished with an error. In such case, the **Path** view can be used for detecting the reasons that led to the error. The view displays the execution history provided by JPF, but in a more convenient way than the command-line program. The execution history, or path in other words, is split into a set of transitions which are listed in a grid located in the left part of the view. The right part shows the detailed information about the selected transition. Each transition starts with so called *choice* and choices are produced by so called *choice generators*. Each generator produces a new choice (and thus a transition) every time JPF backtracks to the state in which the generator was created. Users can see the following data about each transition:

- number of the thread that executed the transition,
- choice generator or type of the choice (e.g., selection from a set of threads to be executed),
- additional information about the choice (e.g., which thread was selected),
- instruction that caused a choice generator to be created,
- number of the choice in the choice generator (in the form of *choice_number* / *choice_total*).

Figure A.4 Path View



Transitions in the grid are coloured differently, those executed by the same thread are coloured with the same background colour. When a transition is selected, all other transitions that were executed by the same thread are highlighted in bold. Users can use either the keyboard arrows or the arrow buttons in top right corner of the view to move in the grid. The arrows marked with T jump only between the transitions executed by the same thread.

Three other view buttons allow to

- show the bytecode instructions along with the source code listing,

- trace the standard Java libraries (can be combined with the previous button),
- search the code listing.

A.3 Ant Task

For better integration of UnitCheck into existing projects, the Ant task is provided. This task should always be preferred to the command-line program because it offers a more convenient way of running JUnit tests under JPF.

After extracting `unitcheck-ant-task-r*.zip` (or `tgz`), the following structure will appear:

- `lib` contains the taks's binary code and its dependencies in the form of JAR files,
- `src` and `classes` contain sample JUnit tests,
- `examples` contain Ant build files that use the UnitCheck task for checking the sample tests.

Before the task can be used, it must be properly defined in a build file as Listing A.1 shows. The task depends on a couple of JAR files located in its `lib` directory. According to the Ant guidelines, this way of defining task dependencies is more flexible than using the Ant global `lib` directory.

Listing A.1: Definition and Usage of the UnitCheck Ant Task

```
<?xml version="1.0" ?>
<project name="myproject" default="mytarget" basedir=".">
  <property name="unitcheck.task.dir"
    location="<unitcheck-ant-task-dir>/lib" />

  <path id="unitcheckcp">
    <fileset dir="${unitcheck.task.dir}" includes="*.jar" />
  </path>

  <taskdef name="unitcheck"
    classname="cz.kebrt.unitcheck.ant.UnitCheckTask"
    classpathref="unitcheckcp"
  />

  <target name="mytarget">
    <property name="proj.dir" location="/project/daisy" />
    <unitcheck basedir="${unitcheck.task.dir}">
      <vmclasspath>
        <pathelement path="${proj.dir}/classes" />
      </vmclasspath>
      <vmsourcepath path="${proj.dir}/src,${proj.dir}/testsrc" />

      <jpfproperty key="jpf.listener"
        value="daisy.jpf.DaisyLockOrderProperty" />
      <jpfproperty key="search.properties"
        value="gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty" />
```

```
      <test class="daisy.unittest.CreatFileTest"
        method="testCreatFile" />
      <test class="daisy.unittest.CreatFileTest"
        method="testCreatLongFilename" />

      <batchtest>
         <fileset dir="${proj.dir}">
           <include name="**/*Test*.java" />
         </fileset>
      </batchtest>
    </unitcheck>
  </target>

</project>
```

The `unitcheck` task has the following attributes and subelements, the ones marked with * are required.

Attribute name	Default	Description
`basedir` *		Path to the `lib` directory of the UnitCheck task. The same directory is used in the task definition.
`vmclasspathref` *		Reference to the classpath that is used when checking the input test classes. The `vmclasspath` element can be used instead.
`vmsourcepathref`		Reference to the sourcepath that is used when reporting errors. If not specified, the execution history might be empty. The `vmsourcepath` element can be used instead.
`conffile`		Path to a JPF configuration file. The default JPF configuration parameters are overriden with the values from this file. The `jpfproperty` elements can be used instead.
`printpath`	`true`	Print the execution history (path) when an error occurs.
`printbytecode`	`false`	Print the bytecode intructions along with the execution history.
`filterjunittrace`	`true`	Filter the JUnit and UnitCheck code out of the execution history.
`tracejre`	`false`	Trace the standard Java libraries in the execution history.

Element name	Description
`vmclasspath` *	Classpath that is used when checking the input test classes. All the standard Ant constructs for creatings path-like structures can be used within the element. The `vmclasspathref` attribute can be used instead.

Element name	**Description**
`vmsourcepath`	Sourcepath that is used when reporting errors. If not specified, the execution history might be empty. All the standard Ant constructs for creatings path-like structures can be used within the element. The `vmsourcepathref` attribute can be used instead.
`jpfproperty`	Allows to override the default JPF configuration parameters. Each property has the `key` and `value` attributes.
`test`	Defines a JUnit test to be checked[4]. Each test must have a test `class` assigned. The `method` attribute allows to specify only a single test method, otherwise all test methods of the class are checked.
`batchtest`	More tests to be checked can be specified by this element. All the standard Ant constructs for defining filesets can be used within the element. Only the `.java` and `.class` files are taken into account.

[4] Remember that the classpath containing the test and its dependencies must be configured using `vmclasspath` or `vmclasspathref`.

Bibliography

[1] Agitar, `http://www.agitar.com`.

[2] Apache ant manual, `http://ant.apache.org/manual`.

[3] Bogor, `http://bogor.projects.cis.ksu.edu`.

[4] CMBC – Bounder Model Checker, `http://www.cprover.org/cmbc`.

[5] CodePro AnalytiX, `http://www.instantiations.com`.

[6] Common Aspect Proof Environment (CAPE), `http://www.cs.technion.ac.il/~ssdl/research/cape`.

[7] CoView, `http://www.codign.com`.

[8] Daisy file system. joint CAV/ISSTA special event on specification, verification, and testing of concurrent software, `http://research.microsoft.com/en-us/people/qadeer`.

[9] Equinox, an implementation of the OSGi R4 core framework, `http://www.eclipse.org/equinox`.

[10] Java PathFinder Eclipse PlugIn (JPFep) project, `https://ssdl-linux.cs.technion.ac.il/trac/JPFep`.

[11] Java PathFinder (JPF), `http://javapathfinder.sourceforge.net`.

[12] JFace UI toolkit, `http://wiki.eclipse.org/index.php/JFace`.

[13] JUnit testing framework, `http://www.junit.org`.

[14] Nebula project – supplemental custom widgets for SWT, `http://www.eclipse.org/nebula`.

[15] OSGi – the dynamic module system for java, `http://www.osgi.org`.

[16] Parasoft Jtest: Java Testing Toolkit, `http://www.parasoft.com`.

[17] Spin, `http://spinroot.com`.

[18] SWT: The standard widget toolkit, `http://www.eclipse.org/swt`.

[19] TLC, `http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html`.

[20] Visual Java PathFinder (VJP), `http://visualjpf.sourceforge.net`.

[21] Z3: SMT Solver, `http://research.microsoft.com/en-us/um/redmond/projects/z3`.

[22] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[23] An eclipse plug-in for model checking. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, page 251, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Paul Ammann, Paul E. Black, and Wei Ding. Model checkers in software testing. Technical report, NIST-IR 6777, National Institute of Standards and Technology, 2002.

[25] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM '98*, pages 46–54. IEEE Computer Society, 1998.

[26] Saswat Anand, Corina Pasareanu, and Willem Visser. JPF–SE: A symbolic execution extension to java pathfinder. In *TACAS*, volume 4424 of *LNCS*, pages 134–138. Springer, 2007.

[27] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.

[28] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using modelchecking, 1996.

[29] Cem Kaner. QAI QUEST Conference, Chicago, April 2008, `http://www.kaner.com`.

[30] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[31] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, September 2004.

[32] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

[33] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison Wesley Professional, 2006.

[34] Adam Kiezun David Harvison, Vijay Ganesh. jFuzz: A concolic whitebox fuzzer for java. In *The First NASA Formal Methods Symposium*, pages 121–125, 2009.

[35] Niels H. M. Aan de Brugh, Viet Yen Nguyen, and Theo C. Ruys. Moonwalker: Verification of .net programs. In *TACAS*, pages 170–173, 2009.

[36] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[37] Elliotte Rusty Harold. An early look at JUnit 4. *IBM developerWorks*, 2005.

[38] Elsa L. Gunter and Doron Peled. Unit Checking: Symbolic Model Checking for a Unit of Code. In *Verification: Theory and Practice*, pages 548–567, 2003.

[39] Fevzi Belli and Baris Güldali. Software Testing via Model Checking. In *ISCIS*, pages 907–916, 2004.

[40] Francesca Arcelli, Claudia Raibulet, Ivano Rigo, and Luigi Ubezio. An Eclipse Plug-in for the Java PathFinder Runtime Verification System. In *SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 142–152, Washington, DC, USA, 2006. IEEE Computer Society.

[41] Gregor Engels and Baris Güldali and Marc Lohmann. Towards Model-Driven Unit Testing. In Thomas Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 182–192. Springer, 2006.

[42] Guillaume P. Brat and Doron Drusinsky and Dimitra Giannakopoulou and Allen Goldberg and Klaus Havelund and Michael R. Lowry and Corina S. Pasareanu and Arnaud Venet and Willem Visser and Richard Washington. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods in System Design*, 25(2-3):167–198, 2004.

[43] Hao Chen and Drew Dean and David Wagner. Model Checking One Million Lines of C Code. In *NDSS*, 2004.

[44] Klaus Havelund. Java PathFinder, A Translator from Java to Promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, page 152, London, UK, 1999. Springer-Verlag.

[45] Klaus Havelund and Willem Visser. Program Model Checking as a New Trend. *STTT*, 4(1):8–20, 2002.

[46] Johannes Link and Peter Frohlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publisher, 2003.

[47] Jonathan de Halleux and Nikolai Tillmann. Parameterized Unit Testing with Pex. *Tests and Proofs*, pages 171–181, 2008.

[48] Jonathan de Halleux, Nikolai Tillmann, Wolfram Schulte. Parameterized Unit Testing with Pex – Tutorial, `http://research.microsoft.com`.

[49] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[50] Lasse Koskela. *Test Driven: Practical TDD and acceptance TDD for Java developers*. Manning, 2008.

[51] Madanlal Musuvathi and David Park and Andy Chou and Dawson Engler and David Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation*, December 2002.

[52] Masoud Mansouri-Samani, Corina S. Pasareanu, John J. Penix, Peter C. Mehlitz, Owen O'Malley, Willem C. Visser, Guillaume P. Brat, Lawrence Z. Markosian, Thomas T. Pressburger. *Program Model Checking – A Practitioner's Guide*. NASA Ames Research Center, 2007.

[53] Michael Ziller. PeX – Parameterized Unit Tests in Visual Studio. *University of Karlsruhe, Formal Software Development Seminar*, 2008.

[54] Michal Kebrt and Ondřej Šerý. UnitCheck: Unit Testing and Model Checking Combined. *Accepted for publication in Proceedings of ATVA'09, LNCS*, 2009.

[55] Mike Barnett and Nikolai Tillmann. Contract Checking and Automated Test Generation With Pex, PDC2008.

[56] Jan Tobias Mühlberg and Gerald Lüttgen. Blasting linux code. In *FMICS/PDMC*, pages 211–226, 2006.

[57] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 267–280. USENIX Association, 2008.

[58] Oksana Tkachuk and Matthew B. Dwyer and Corina S. Pasareanu. Automated Environment Generation for Software Model Checking. In *ASE*, pages 116–129, 2003.

[59] Gerrit Rothmaier, Tobias Kneiphoff, Heiko Krumm, and Bosch Rexroth Ag Witten. Using spin and eclipse for optimized high-level modeling and analysis of computer network attack models, 2005.

[60] Scott Ranville and Paul E. Black. Automated Testing Requirements-Automotive Perspective. *The Second International Workshop on Automated Program Analysis, Testing and Verification*, 2001.

[61] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. pages 419–423. 2006.

[62] Tillmann, Nikolai and Schulte, Wolfram. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, 2005.

[63] Aleksander Vreže Tim Kovše, Boštjan Vlaovič and Zmago Brezočnik. Eclipse plug-in for spin and st2msc tools-tool presentation. In *Model Checking Software*, volume 5578/2009 of *Lecture Notes in Computer Science*, pages 143–147. Springer, 2009.

[64] Vincent Massol and Ted Husted. *JUnit in Action*. Manning, 2004.

[65] Willem Visser, Klaus Havelund, and Guillaume Brat. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12, 2000.

[66] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

[67] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, Edinburgh, UK, April 2005.

[68] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.

[69] Zhu, Hong and Hall, Patrick A. V. and May, John H. R. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.